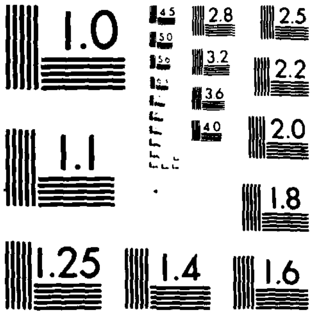
AD-A109 979   TEXAS INSTRUMENTS INC LEWISVILLE EQUIPMENT GROUP   F/G 9/2
ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPE--ETC(U)
DEC 81   F30602-80-C-0293
UNCLASSIFIED   RADC-TR-81-360-VOL-4   NL

1 of 1
AD
A109979

END
DATE
FILMED
3-82
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

PHOTOGRAPH THIS SHEET

DTIC ACCESSION NUMBER

AD A109979

LEVEL

INVENTORY

I

Texas Instruments, Inc
Lewisville, TX Equipment Group - ACSL
ADA Integrated Environment III Computer Program
Development Specification. Interim Rpt. 15 Sep. 80 - 15 Mar. 81
Dec. 81
Contract F30602-80-C-0293 RADC-TR-81-360, Vol. IV

DOCUMENT IDENTIFICATION

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR		
NTIS	GRA&I	☒
DTIC	TAB	□
UNANNOUNCED		□
JUSTIFICATION		
BY		
DISTRIBUTION /		
AVAILABILITY CODES		
DIST	AVAIL AND/OR SPECIAL	
A		

DISTRIBUTION STAMP

DTIC
COPY
INSPECTED
3

DTIC
S ELECTE D
JAN 25 1982
D

DATE ACCESSIONED

82 01 12 007

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

RADC-TR-81-360, Vol IV (of four)
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

Texas Instruments, Inc.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-360, Volume IV (of four) has been reviewed and is approved for publication.

APPROVED: Elizabeth S. Kean
ELIZABETH S. KEAN
Project Engineer

APPROVED:
JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss
JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, pleaee notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-360, Vol IV (of four)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ADA INTEGRATED ENVIRONMENT III COMPUTER PROGRAM DEVELOPMENT SPECIFICATION		5. TYPE OF REPORT & PERIOD COVERED Interim Report 15 Sep 80 - 15 Mar 81
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0293
9. PERFORMING ORGANIZATION NAME AND ADDRESS Texas Instruments Incorporated Equipment Group-ACSL, P O Box 405, M.S. 3407 Lewisville TX 75067		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62204F/33126F/62702F 55811919
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441		12. REPORT DATE December 1981
		13. NUMBER OF PAGES 80
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Elizabeth S. Kean (COES)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada
Compiler
Database
KAPSE
MAPSE
Kernel
Debugger
APSE
AIE
Integrated environment
Editor

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This B-5 Specification describes, in detail, the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an

APSE is built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this specification include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

TABLE of CONTENTS

Paragraph Title Page

SECTION 1 SCOPE

SECTION 2 APPLICABLE DOCUMENTS

SECTION 3 REQUIREMENTS

SECTION 4 QUALITY ASSURANCE PROVISIONS

LIST of FIGURES

LIST of EXAMPLES

SECTION 1

SCOPE

1.1 Identification

This specification establishes the requirements for the performance, design, test and acceptance of a computer program configuration item identified as the Ada Programming Toolset, CPCI C04. This CPCI contains the following computer programs for use in Ada software design, development, maintenance and management activities.

- C04.01 General Text Editor
- C04.02 Ada Program Binder
- C04.03 Ada Source-Level Debugger

These tools are provided on all computer systems supporting the Ada Integrated Environment.

1.2 Functional Summary

The purpose of this specification is:

1. To identify the functional capabilities of each software tool
2. To describe the interfaces between each tool and the Ada Software Environment, other Ada programs, the database, and users.

All programs in the toolset are written to be as transportable and machine relative as is practically possible.

SECTION 2
APPLICABLE DOCUMENTS

The following documents form a part of this specification to the extent specified herein. Unless stated otherwise the issue in effect on the date of this specification shall apply.

2.1 Program Definition Documents

[DoD80A] Requirements for Ada Programming Support Environments: "STONEMAN", DoD (February 1980).

[RADC80] Revised Statement of Work for Ada Integrated Environments, RADC, Griffiss Air Force Base, NY (March 1980).

[SOFT80A] Ada Compiler Validation Capability: Long Range Plan, SofTech Inc., Waltham, MA (February 1980).

[SOFT80B] Draft Ada Compiler Validation Implementers' Guide, SofTech Inc., Waltham, MA (October 1980).

2.2 Military Specifications and Standards

[DoD80B] Reference Manual for the Ada Programming Language: Proposed Standard Document, DoD (July 1980) (reprinted November 1980).

SECTION 3

REQUIREMENTS

3.1 Introduction

This part of this specification gives a general description, describes interfaces, and provides detailed functional requirements for the Ada Software Toolset text editor, Ada program binder, and source language level debugger.

3.1.1 General Description

The software toolset contains system-provided or user-written programs used in software design, development, maintenance and management activities. The software tools included in the minimal Ada Programming Support Environment and described in this specification are:

- An interactive general purpose text editor, providing facilities for the creation and modification of text files containing Ada source text or other programs, data or documentary material. The editor includes interfaces to database and configuration management tools.
- A program binder, to integrate Ada program units into complete programs and map the resulting programs from the Ada virtual machine to the architecture and configuration of a target machine.
- An interactive, source-language-level debugger, providing diagnostic facilities for analysis of runtime errors in programs executing on the Ada Integrated Environment host computer or on an appropriately connected target machine.

3.1.2 Program Interfaces

The components of the software toolset are all Ada programs. Each communicates with its environment by assignment of values to parameters at program invocation, or by the use of standard subprograms and packages that interface with the underlying operating system to provide requested services, such as:

* Logical input/output

* Dynamic storage allocation

* Inquiries concerning system status

* Values of program, task or file attributes.

A logical input/output package is provided to implement a uniform, device-independent approach to data transfer. Logical input/output capabilities are provided for database objects, input/output devices, interactive terminals, and inter-program communications.

A standard package provides controlled facilities to interrogate or modify the values of the user-maintained attributes of database objects and relations between them. This package is integrated with the logical input/output package to provide a uniform treatment of the attributes of all logical files.

The general interface between the user and each running program is a control file containing detailed instructions to specify the processing to be performed by the program. Each tool in the Ada Integrated Environment shall incorporate standard provisions for control input from an interactive terminal or from a file stored in the database. In certain cases, each tool may implement a specialized control language.

3.2 Detailed Functional Requirements

3.2.1 Ada Interactive Editor

The Ada Interactive Text Editor provides a user with the capability to create and modify files of textual data, whether the data is Ada source text or documentary material.

The Editor may be classified as a "display-oriented context editor." It takes full advantage of the capabilities of state of the art video display terminals. It uses the editing capabilities of video display terminals such as character insertion/deletion to minimize the processing required by the host computer for these functions.

Although the Editor is designed to be used at a display terminal, it is also possible to use it from a line-at-a-time device such as a teletypewriter or a file of commands.

3.2.1.1 Program Interfaces

The Editor is an Ada program. It uses the facilities of the Database Subsystem to create, write and read temporary files, to save created files, and to access existing files. The editor invokes the archiving utilities to save new revisions to text files.

3.2.1.2 User Interface

The Editor's interface with an interactive user is defined by the Editor Command Language and by terminal-dependent packages which adapt editor functions to the keyboard and display capabilities of teletypewriters and various forms of video display terminals.

3.2.1.2.1 Syntax Description

The syntax of the Editor Command Language is described using a variant of Backus-Normal form.

1. Lower case words denote syntactic categories.

2. Upper case words denote keywords in the language.

3. Square brackets enclose optional items from which a choice may be made, if desired.

4. Parentheses enclose required items from which a choice must be made.

5. A vertical bar separates alternative items.

The command language consists of keywords, operators, and a small number of syntactic entities. The syntactic entities used in the language description are as follow.

1. character -- a single ASCII character.

2. number -- a decimal integer.

3. string -- a sequence of one or more characters prefixed and terminated by the string bracket character ("). A string bracket character is represented within a string by doubling it.

4. pointer -- indicates a unit within the text being edited. The size of a unit referred to by a pointer is controlled by the UNIT option of the SET command. The syntax of a pointer is as follows:

```
pointer      ::= . [identifier] [(+|-) number]
```

where the identifier is an Ada identifier.

3.2.1.2.2 Command Summary

The following is a summary of the editor commands.

```
Edit Session Termination

      ABORT
      QUIT    ([filename] [REPLACE]|ABORT)
      SAVE    [filename] [REPLACE]
      SUSPEND

Data Transfers Between Files

      READ    filename [FROM pointer1 [THRU pointer2]] [TO pointer3]
      WRITE   [pointer1] [THRU pointer2] TO filename [APPEND]

Pointer Movement

      BACKWARD number
      BOTTOM
      FIND     string [THRU pointer]
      FORWARD  number
      TO       pointer
      TOP
```

Data Modification

```
CHANGE  string1 TO string2 [THRU pointer] [QUERY]
APPEND  string [TO pointer]
DELETE  pointer1 THRU pointer2
INSERT  string [AT pointer]
REPLACE [pointer] [WITH] string
RESTORE
RESUME
```

Data Movement

```
COPY    pointer1 [THRU pointer2] [TO pointer3]
INDENT  [pointer1] [THRU pointer2] FOR number
INPUT
JOIN
MOVE    pointer1 THRU pointer2 TO pointer3
SPLIT   [pointer|column_number|string]
```

Miscellaneous Commands

```
DISPLAY pointer1 THRU pointer2
HELP    [command_name]
NAME    pointer
REDO    number
SET     options -- The available options are as follow:
    CASE     (UPPER|MIXED)
    CURSOR   (UNDERSCORE|BRACKET character1 character2)
    KEY      string
    POSITION (BEFORE|AFTER)
    RANGE    LINES [pointer1] [THRU pointer2]
    RANGE    COLUMNS [first_column] [THRU last_column]
    SCREEN   ROWS number_of_rows
    SEARCH   (BEGIN|END)
    SEARCH   (GENERAL|EXACT)
    SPAN     number
    SYNONYM  (OFF|ON|newname oldname)
    TAB      (WORD string|COLUMNAR string)
    TERMINAL (TELETYPEWRITER|IBM3270|ASCII_VDT)
    TRUNCATE (ON|OFF)
    UCHAR    (OFF|character)
    UNIT     (LINE|CHARACTER)
    VERIFY   (ON|OFF)
SHOW (FILENAME|OPTIONS|POINTER|SIZE)
```

Editor commands may be entered in either upper- or lower-case letters and may be abbreviated to the minimum number of characters necessary for it to be recognized.

A unit is either a text line or a character, as indicated by the SET UNIT command. It is the entity manipulated by the commands of the editor.

The following pointer names are predefined in the editor and may not be changed by the user:

* ".CURRENT" or ".", which indicates the current text location of the text being edited;

* ".TOP", which indicates the first unit of the text being edited;

* ".BOTTOM", which indicates the last unit of the text being edited;

* ".FIRST", which indicates the first unit of the range being edited [see SET RANGE command];

* ".LAST", which indicates the last unit of the range being edited [see SET RANGE command];

* ".LEFT", which indicates the first character of the current line being edited; or

* ".RIGHT", which indicates the last character of the current line being edited.

3.2.1.3 Editor Commands

3.2.1.3.1 ABORT Command

```
ABORT
```

The ABORT command terminates an edit session without saving the editing done during the session.

3.2.1.3.2 APPEND Command

```
APPEND string [TO pointer]
```

The APPEND command appends the specified string to the end of the line specified by "pointer" (default ".CURRENT").

3.2.1.3.3 BACKWARD Command

```
BACKWARD number
```

The BACKWARD command positions the current text pointer a specified number of lines from the current line toward the top of the text.

3.2.1.3.4 BOTTOM Command

BOTTOM

The BOTTOM command positions the current text pointer to the bottom of the text (i.e., the last unit of the text).

3.2.1.3.5 CHANGE Command

CHANGE string1 TO string2 [THRU pointer] [QUERY]

The CHANGE command changes the first occurrence of string1, searching in the direction of "pointer" (default ".BOTTOM"), to string2. If string1 is found between the current text location and the text location indicated by "pointer", the current text pointer is modified to refer to the location of the located string, otherwise the current text pointer is not modified.

If the QUERY option is specified then each line which can be modified by the command is displayed in its modified form and the user is prompted as to whether the modification should take place or not. Following are the permitted responses to the prompt.

- YES -- the modification should occur.
- NO -- the modification should not occur.
- ALL -- verification is turned off for the rest of the search.
- ABORT -- terminate the command.

3.2.1.3.6 COPY Command

COPY pointer1 [THRU pointer2] [TO pointer3]

The COPY command duplicates the text between "pointer1" and "pointer2" (default "pointer1") at the text location indicated by "pointer3" (default ".CURRENT").

3.2.1.3.7 DELETE Command

DELETE pointer1 [THRU pointer2]

The DELETE command deletes the text between "pointer1" (default ".CURRENT") and "pointer2" (default ".CURRENT"). The unit following "pointer2" becomes the current text location.

3.2.1.3.8 DISPLAY Command

DISPLAY pointer1 THRU pointer2

The DISPLAY command displays a specified portion of the text being edited. This command is intended, primarily, for use on teletypewriters.

3.2.1.3.9 FIND Command

FIND string [THRU pointer]

The FIND command locates a string in the text being edited starting at the unit immediately following or preceding the current text pointer and moving in the direction of "pointer" (default ".BOTTOM"). If "string" is not found between the current text position and "pointer", the current text pointer is not changed.

3.2.1.3.10 FORWARD Command

FORWARD number

The FORWARD command positions the current text pointer a specified number of lines from the current line toward the bottom of the text.

3.2.1.3.11 HELP Command

HELP [command_name]

The HELP command assists the user in the use of the Ada Editor. The parameter is the name of an Ada Editor command. The response is a description of the specified command.

3.2.1.3.12 INDENT Command

INDENT [pointer1] [THRU pointer2] AT column_number

The INDENT command specifies that the first nonblank character on each line within the specified range is to occur in the specified column_number. This command is provided to assist the user in the formatting of program text.

3.2.1.3.13 INPUT Command

INPUT

The INPUT command places the editor into input mode. The user may then insert text at the current text location. The input mode is terminated by pressing the input termination key for the terminal being used.

3.2.1.3.14 INSERT command

INSERT string [AT pointer]

The INSERT command inserts a string at the text location indicated by "pointer" (default ".CURRENT").

3.2.1.3.15 JOIN Command

JOIN

The JOIN command causes the line following the current line to be appended to the current line such that the first nonblank character of the following line is separated from the last nonblank character of the current line by a blank. The position of the current text pointer is not changed.

3.2.1.3.16 MOVE Command

MOVE pointer1 [THRU pointer2] [TO pointer3]

The MOVE command moves the text between "pointer1" and "pointer2" (default "pointer1") to the text location indicated by "pointer3" (default ".CURRENT")

3.2.1.3.17 NAME Command

NAME pointer

The NAME command gives a "name" to the current text unit in the text being edited. This command enabies a user to place a "paperclip" at a specified text location, which may be referenced later in any of the commands that require a pointer.

3.2.1.3.18 QUIT Command

QUIT ([filename] [REPLACE]|ABORT)

The QUIT command terminates an edit session and saves the results of the session in the specified file (default is the associated filename). If the external file specified already exists, the REPLACE option must be used to replace it. The ABORT parameter indicates that the session is to be terminated without saving the results of the editing session (i.e., the same as the ABORT command).

3.2.1.3.19 READ Command

READ filename [FROM pointer1 [THRU pointer2]] [TO pointer3]

The READ command copies the text lines between "pointer1" (default ".TOP") and "pointer2" (default ".BOTTOM") of the external file specified to the text location indicated by "pointer3" (default ".CURRENT") within the text being edited.

3.2.1.3.20 REDO Command

REDO number

The REDO command repeats the last command entered a specified number of times. The commands which may be repeated are FIND, CHANGE, COPY, and INSERT.

3.2.1.3.21 REPLACE Command

REPLACE [pointer] [WITH] string

The REPLACE command replaces the text indicated by "pointer" (default ".CURRENT") with the text in the string.

3.2.1.3.22 RESTORE Command

RESTORE

The RESTORE command restores the last portion of text that was deleted. The text is inserted at the current text location of the edited text.

3.2.1.3.23 RESUME Command

RESUME

The RESUME command causes the editor to resume the last edit session at the state in which it was suspended. It may only be specified as the first command upon entry to the editor.

3.2.1.3.24 SAVE Command

SAVE [filename] [REPLACE]

The SAVE command saves the results of the edit session in the specified external file (default is the associated filename). If the external file specified already exists, the REPLACE option must be used to replace it.

The current text location is not affected by this command and the edit session resumes at the same state it was in before the command was entered.

3.2.1.3.25 SET Command

SET options

- SET CASE (UPPER|MIXED)

  - indicates whether the text entered by the user is to be converted to upper case (UPPER) or not modified (MIXED, the default) when entered.

- SET CURSOR (UNDERSCORE|BRACKET character1 character2)

  - designates the method by which the current text location is indicated when the TERMINAL option of the SET command is TELETYPEWRITER. "BRACKET character1 character2" indicates that the current text location is to be bracketed by the characters "character1" and "character2". UNDERSCORE specifies that the current text location is to be indicated by underlining it (This only works if the terminal being used has the backspace capability). The default is "BRACKET [ ]".

- SET KEY string

  - defines the meaning of the specified command key, or removes any meaning associated with the specified command key. This command is terminated by entry of one of the command keys defined for the terminal being used, rather than the usual entry terminator. The parameter "string" is any valid editor command. As a result of this command, any time that the command key which is used as a terminator is pressed, "string" is interpreted as a command to the editor.

- SET POSITION (BEFORE|AFTER)

  - indicates whether text is to be inserted before or after the target string when using the INSERT, APPEND, MOVE, READ, and COPY commands.

- SET RANGE LINES pointer1 THRU pointer2

  - indicates the range of lines of text in which editing commands may have an effect. This option modifies ".FIRST" to refer to the same text location as "pointer1" and ".LAST" to refer to the same text location as "pointer2."

- SET RANGE COLUMNS [first_column] [THRU last_column]

  - indicates the range of columns of text in which editing commands may have an effect. This option modifies ".LEFT" to refer to the first_column of the current text line and ".RIGHT" to refer to the last_column of the current text line.

* SET SCREEN ROWS number_of_rows

- indicates the number of rows which the editor should use. The default is the number of rows on the screen.

* SET SEARCH (BEGIN|END)

- indicates that the current text pointer is to refer to the first character (BEGIN, the default) or last character (END) of a string searched for in a FIND or CHANGE command, if the command is successful.

* SET SEARCH (GENERAL|EXACT)

- indicates that either case (upper or lower) is important (EXACT) or unimportant (GENERAL, the default) in the first string of a FIND or CHANGE command.

* SET SPAN number

- indicates that the first string in a FIND or CHANGE command may span "number" lines of text (default 1).

* SET SYNONYM (OFF|ON|newname oldname)

- indicates that synonym substitution should (ON) or should not (OFF, the default) be used for the recognition of commands; or defines a synonym (newname) for an existing command (oldname).

* SET TAB (WORD string|COLUMNAR string)

- indicates how the TAB key on the user's terminal is to function (if the user's terminal permits such a specification). WORD indicates that the tab positions within a line should be dynamically based upon the contents of the line such that each tab position is at the beginning of a word, where a word is a string of characters delimited by the characters specified in the given string. COLUMNAR indicates that the tab positions should be based upon a list of columns, where the columns are specified as a list of numbers separated by commas in the given string. This option affects only those terminals which transmit a TAB character. There is no default for this command; pressing the TAB key has no function unless this option is set.

* SET TERMINAL (TELETYPEWRITER|IBM3270|ASCII_VDT)

- indicates the type of terminal being used during the edit session. It also enables a user to specify that the editor should function as if in TELETYPEWRITER mode when using a video display terminal.

- SET TRUNCATE (ON|OFF)

  - indicates whether a line of text, which is too long to be displayed on one row should be truncated(ON, the default) or displayed on several rows(OFF).

- SET UCHAR (OFF|character)

  - sets the universal character to the given character (character) or turns off this capability (OFF, the default). When the universal character is used in a FIND or CHANGE command it represents a don't care string. For example, if the universal character is $, the command "FIND animal$house" could possibly locate any of the strings:

    ... animal house,

    ... animals in the house, or

    ... animals should not be in the house.

- SET UNIT (LINE|CHARACTER)

  - indicates whether pointers refer to lines of text(LINE, the default) or characters within the text(CHARACTER).

- SET VERIFY (ON|OFF)

  - indicates whether the current line is to be displayed when the current text pointer changes locations. This command is principally for use with teletypewriters.

3.2.1.3.26 SHOW Command

SHOW (FILENAME|OPTIONS|POINTER|SIZE)

- SHOW FILENAME

  - Displays the name of the external file associated with the text being edited.

- SHOW OPTIONS

  - Displays the options controlled by the SET command.

- SHOW POINTER pointer

  - Displays the line number and column number of "pointer."

- SHOW SIZE

  - Displays the number of lines in the text being edited.

3.2.1.3.27 SPLIT Command

SPLIT [column_number|string]

The SPLIT command splits the current text line into two lines. If the UNIT option of the SET command is LINE, the column in which the split is to occur must be specified. The POSITION option of the SET command determines whether the split occurs before or after the specified column number or string. If the UNIT option of the SET command is CHARACTER, the split occurs at the current text location.

3.2.1.3.28 SUSPEND Command

SUSPEND

The SUSPEND command terminates an edit session which the user desires to resume at some later time. This command causes the state of the current edit session to be saved and the edit session to terminate. The external file associated with the text being edited is not modified.

3.2.1.3.29 TO Command

TO pointer

The TO command positions the current text pointer to the text location indicated by "pointer."

3.2.1.3.30 TOP Command

TOP

The TOP command positions the current text pointer to the top of the text (i.e., the first unit of the text).

3.2.1.3.31 WRITE Command

WRITE pointer1 [THRU pointer2] TO filename [APPEND]

The WRITE command places the text between "pointer1" and "pointer2" (default "pointer1") into the external file specified by filename. If the APPEND option is specified, the text is added to the end of the specified external file, otherwise the external file is replaced.

3.2.1.4 Function Descriptions

The Editor consists of four packages of procedures and functions. The packages divide the labor of editing along the lines of functionality in order to make modification of the editor a simple task.

* The Presentation Manager handles communications with the terminal to which the editor is interfacing. Because of the many types of terminals and their many capabilities this portion of the editor

handles all terminal dependent communications. These communications may be as complicated as the management of character level input/output from a device such as a dumb video display terminal or as simple as transmitting and receiving text one line at a time such as from a teletypewriter.

* The Edit Command Interpreter translates editor commands entered by a user into a sequence of actions required by the editor. The commands are strings received from the Presentation Manager which are translated into calls to procedures in the Edit Command Processor.

* The Edit Command Processor performs the majority of the operations that are required of the editor. The Edit Command Processor performs many of the operations by itself and translates others to calls to routines in the Edit List Manager.

* The Edit List Manager manages the internal representation of a text file while it is being edited.

3.2.1.5 Processing Requirements

3.2.1.5.1 Presentation Manager

The Presentation Manager manages the editors input/output processing with teletypewriters or video displays.

3.2.1.5.1.1 Teletypewriter Interface

The teletypewriter terminal interface is relatively simple. Its communications are character oriented and require the Presentation Manager to perform only the following operations:

* Transmit a character string to the terminal,

* Receive a character string from the terminal, and

* Intercept nonprintable interrupt keys pressed by the user.

For the majority of the time, the Presentation Manager will be transmitting a character string which is the current text line or the output from a command.

The only form of input entered by a user at a teletypewriter is an edit command or a special key which must be translated by the Presentation Manager into a character string to be passed to the Edit Command Interpreter.

3.2.1.5.1.2 Video Interface

The video display terminal interface is much more complicated because of the many different kinds of displays on the market. The displays differ in many respects. Some of the differences between terminals are screen size (number of rows and columns), nonprintable characters available, modes of operation available (character, line, or screen), special keys (line insert/delete and character insert/delete), function keys, capability to protect fields on the screen, intelligence (dumb, smart, intelligent), options for intensity, blink, and reverse video, and many others.

The goal of the Editor's video interface design is to use as many of each terminals capabilities as possible, yet maintain the editor's transportability between different terminals.

3.2.1.5.2 Edit Command Interpreter

The Edit Command Interpreter translates commands and special keys as entered by a user into a sequence of procedure calls to the Edit Command Processor.

This separation of functionality allows the syntax of the editor command language to be modified easily as new developments are made in the science of text editing.

3.2.1.5.3 Edit Command Processor

The Edit Command Processor is responsible for performing the actions requested by a user. For each command that a user can enter, there is a routine in the Edit Command Processor which performs the operations specified by that command.

3.2.1.5.4 Edit List Manager

The Edit List Manager is a package of procedures and functions which provides a set of primitives for use by the Ada Editor in the creation and modification of text files.

An edit_list is a declared object which is (usually) associated with an external file containing text. The operations which may be performed on an edit_list are similar to those available for general input/output.

Elements of edit_lists may be fetched or changed, as well as inserted and deleted. The element pointer may be set to a specified position. In addition, the current position of the element pointer and the number of elements in the edit_list may be obtained.

An edit_list consists of an unbounded sequence of elements, all of type string. A positive integer number is associated with each element of the edit_list indicating its (ordinal) position number in this sequence.

The operations available for edit_list association with external files are:

procedure ASSOCIATE(LIST: in out EDIT_LIST;NAME: in STRING);

This procedure associates an edit_list with the contents of an existing external file having the specified name. Modifications made to an edit_list do not affect the external file with which it is associated. This procedure uses the procedure OPEN in the package INPUT_OUTPUT and hence, may result in one of the exceptions defined for the OPEN procedure being raised.

After processing has been completed on an edit_list, the contents of the edit_list may be associated with an external file by the ARCHIVE procedure.

procedure ARCHIVE(LIST: in out EDIT_LIST; NAME: in STRING) ;

Establishes a new external file, if necessary, and transfers the information from the edit_list to the external file with the given name. Accesses to an external file may raise the any of the exceptions defined in the package INPUT_OUTPUT, such as NAME_ERROR, STATUS_ERROR, USE_ERROR, DATA_ERROR or DEVICE_ERROR. This procedure uses the facilities provided by the Database Archive Manager to save the edit_list as the new revision of the external file with the given name.

function ASSOCIATED(LIST: in EDIT_LIST) return BOOLEAN;

Returns TRUE, if the edit_list is associated with an external file, FALSE, otherwise.

function NAME(LIST: in EDIT_LIST) return STRING;

Returns the string representing the name of the external file with which the given edit_list is associated. If the edit_list is not associated with an external file, the exception STATUS_ERROR is raised.

procedure DELETE(LIST: in out EDIT_LIST);

Deletes the edit_list, but does not affect the associated external file, if there is one.

3.2.1.5.4.1 Edit List Processing

Elements of an edit_list can be fetched, changed, inserted, or deleted. Each edit_list has a current element position, which is the position number of the element available for the next fetch, change, insert, or delete operation. The current element position can be changed. Positions in an edit_list are expressed in the integer type LIST_INDEX.

An edit_list has a current size, which is the number of elements in the edit_list. When an edit_list is elaborated, it has zero elements and the current element position is set to 1.

The operations available for edit_list processing are described below.

procedure FETCH(LIST: in EDIT_LIST; ELEMENT: out STRING);

Returns, in the ELEMENT parameter, the value of the element at the current position of the given edit_list. The current element position remains the same. The exception END ERROR is raised if the current element position is greater than the end position of the edit_list.

procedure CHANGE(LIST: in EDIT_LIST; ELEMENT: in STRING);

Gives the specified value to the element in the current position of the given edit_list. The current element position remains the same and the number of elements in the edit_list is not modified. The END_ERROR exception is raised if the current element position is greater than the end position.

procedure REMOVE(LIST: in EDIT_LIST);

Removes the element at the current element position from the edit_list. The size of the edit_list is decreased by one.

procedure INSERT(LIST: in EDIT_LIST; ELEMENT: in STRING);

Inserts the specified element into the given edit_list in front of the current element position of the edit_list. The newly inserted element becomes the new current element position of the edit_list. If the current element position is greater than the end position, the given element is inserted into the element position which is one greater than the end element position. The position of the newly inserted element becomes the new current element position and the size of the edit_list is increased by one.

procedure SET_NEXT(LIST: in EDIT_LIST; TO: in LIST_INDEX);

Sets the current element position of the given edit_list to the specified index value (The specified value may exceed the end position).

function SIZE(LIST: in EDIT_LIST) return LIST_INDEX;

Returns the current size of the edit_list.

function NEXT(LIST: in EDIT_LIST) return LIST_INDEX;

Returns the current element position of the edit_list.

3.2.1.5.4.2 Package EDIT_LIST_MANAGER

The specification of the package EDIT_LIST_MANAGER is given below. It provides the calling conventions for the operations described previously in this section.

```
package EDIT_LIST_MANAGER is
  type EDIT_LIST is limited private

  type LIST_INDEX is range 0..implementation_defined

  -- general operations for edit_list manipulation

procedure ASSOCIATE(LIST: in EDIT_LIST; NAME: in STRING);

procedure ARCHIVE(LIST: in EDIT_LIST; NAME: in STRING);

procedure DELETE(LIST: in EDIT_LIST);

function ASSOCIATED(LIST: in EDIT_LIST) return BOOLEAN;

function NAME(LIST: in EDIT_LIST) return STRING;

procedure FETCH(LIST: in EDIT_LIST;ELEMENT: out STRING);

procedure CHANGE(LIST: in EDIT_LIST; ELEMENT: in STRING);

procedure REMOVE(LIST: in EDIT_LIST);

procedure INSERT(LIST: in EDIT_LIST; ELEMENT: in STRING);

procedure SET_NEXT(LIST: in EDIT_LIST; TO: in LIST_INDEX);

function SIZE(LIST: in EDIT_LIST) return LIST_INDEX;

function NEXT(LIST: in EDIT_LIST) return LIST_INDEX;

  -- exceptions that can be raised

  NAME_ERROR   : exception;
  USE_ERROR    : exception;
  STATUS_ERROR : exception;
  DATA_ERROR   : exception;
  DEVICE_ERROR : exception;
  END_ERROR    : exception;

private
  -- declarations of the edit_list private types
end EDIT_LIST_MANAGER;
```

3.2.2 Ada Program Binder

The Ada Program Binder provides the user with facilities to construct executable programs by binding together compiled Ada program units to form program segments and by combining program segments into a complete self-contained program. Capabilities are provided for user specification of memory partitions and overlays, for inclusion of system library routines, and for the construction of data structures needed by the bound program to interface with the command language interpreter, database subsystem and other APSE or target machine functions.

3.2.2.1 Program Interfaces

The Binder interfaces with its users through a control file, which specifies the program to be bound and gives other processing instructions, and through the Ada program library file, which contains information on all necessary constituent units of the program to be bound. The Binder produces database objects containing the bound program, listings and information needed by the debugger and others.

3.2.2.1.1 Library Utility

The Library Utility is the interface between the Binder and the program library file. The library file gives the structure of the program, the status of all compiled program units, and the relationships between program units. The Binder also uses the Library Utility to record the location of bound segments and programs.

3.2.2.1.2 Execution Environment

The Binder interfaces to the KAPSE Execution Environment to provide data needed to load and execute programs. The Binder either builds the subprogram reference tables that are used by the subprogram linkage handlers or prepares tables from which they are built by the loader function of the KAPSE.

3.2.2.1.3 Program Manager

The Binder interfaces to the Program Manager during the program binding phase to reserve memory for the bound program when the Load-and-Go mode of the program binder is in use, or disk space for the storage of the program image when in the Program Image mode.

3.2.2.1.4 Memory Manager

The Binder interfaces to the Memory Manager to provide the Binder with blocks of dynamic memory while creating bound segments and programs.

3.2.2.1.5 Interactive Debugger

The Binder interfaces to the Interactive Debugger through the program binding map, produced by the program binding phase of the Binder. The program binding map provides the Interactive Debugger with program entry displacements, paging and overlay information and bound segment addresses.

3.2.2.2 Functional Descriptions

The Ada Program Binder is composed of two phases. In the first phase, individual program units are gathered into groups, called 'segments'. These segments are the smallest entities to be used for the building of complete programs. A segment may be used in the building of many programs. In the second phase, the program binder organizes bound segments into executable programs. The user is able to control the organization and building of both the segments and the programs.

3.2.2.2.1 Segment Binding

Tne segment binder is the first step in the transformation of object modules produced by the compiler into executable programs which will belong to a system. The segment binder is responsible for the formation of program segments. Program segments, being the smallest entities which may be explicitly included into a program, are planned compositions of named program units. Program segments are grouped with other program segments to form programs. Segments are capable of being shared among multiple programs and a well organized segment may be useful in many executable programs. To provide this capability, the user is given adequate controls to include selected program units, to set segment partition sizes and to determine overlay structures. These controls are effected through a set of commands which direct the structure and composition of the bound segment. The result of the segment binding is an object code module of four distinct parts:

1. Pre-load Code Section Dictionary

2. Loadable Code section

3. Constant Section Dictionary

4. Constant Section

The relationship of these sections is illustrated in Figure 3-1. Operational use of these segments will be discussed in later sections and illustrated in Figure 3-9, Figure 3-10 and Figure 3-11.

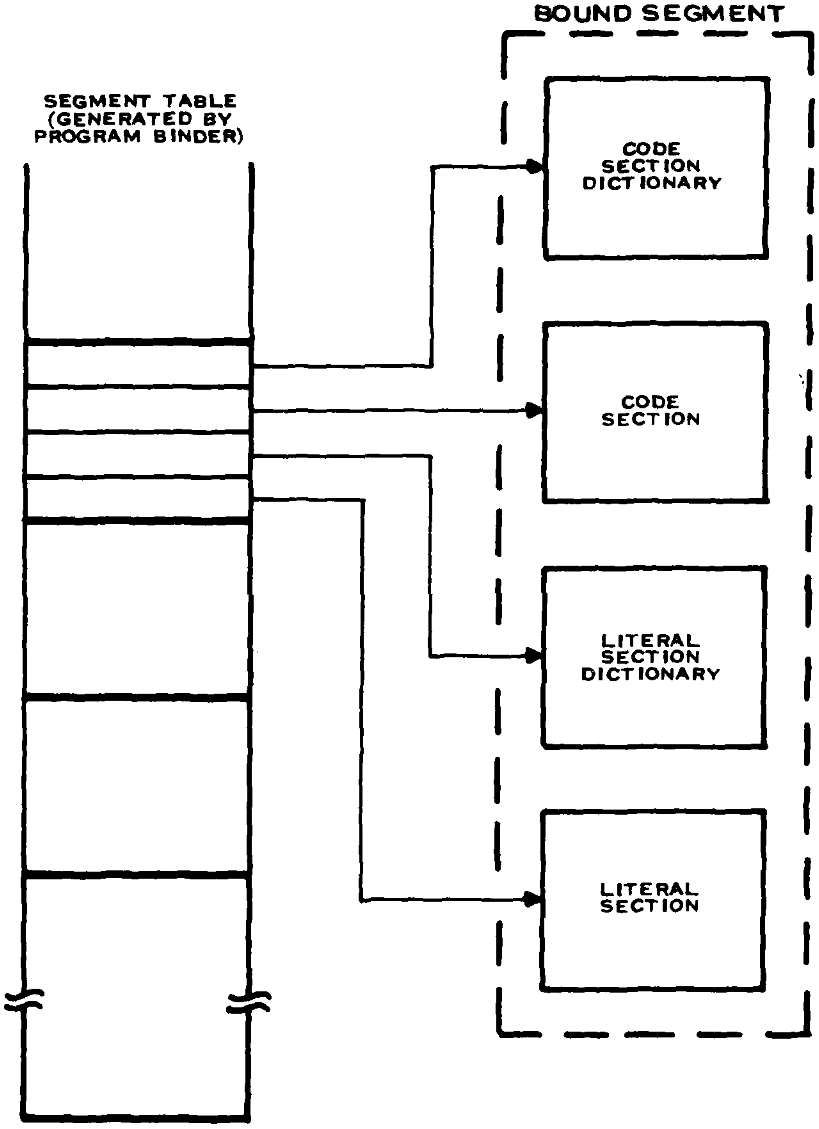

BOUND SEGMENT
SEGMENT TABLE (GENERATED BY PROGRAM BINDER)
CODE SECTION DICTIONARY
CODE SECTION
LITERAL SECTION DICTIONARY
LITERAL SECTION

Figure 3-1 Segment Organizational Parts

3.2.2.2.2 Program Binding

The program binder generates program units which are suitable for target machine execution. The program binder bases the building of the Ada program upon a user designated "main" program segment. External references of this segment is the subject of searches through the pre-load dictionaries of other segments. When an external reference is found, the segment containing the reference is attached to the main segment. References to external subprograms, et.al., from these attached segments is resolved in a similar manner until all external references are resolved.

This binding process involves the construction of many tables and threads which become quite interwoven through the executable environment. The program binder is responsible for the construction of these data structures. These structures will be:

1. The Segment Table
2. The Segment Code Dictionaries
3. The Segment Code Sections
4. The Segment Constant Dictionaries
5. The Segment Constant Sections

3.2.2.3 Processing Requirements

The following sections describe the processing requirements for the Ada Program Binder. The two major functions of the binder are segment binding and program binding. Special requirements are also presented.

3.2.2.3.1 Segment Binding

The segment binder is responsible for the formation of program segments. Figure 3-2 illustrates the overview of the processing which takes place during this phase of the AIE Binder.

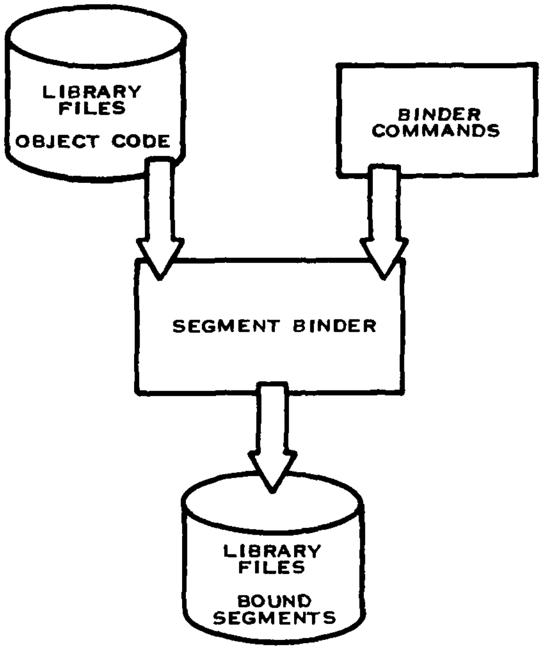


Figure 3-2 Segment Binder Overview

3.2.2.3.1.1 Inputs

The segment binder requires two sources of input for its proper operation: user commands and program units. The user commands which determine the structure and composition of the bound segments. These commands are available to the binder through the KAPSE, but normally originate through a user terminal or control file in the library. The commands to the binder are:

1. INCLUDE -- The INCLUDE command designates a specific program unit for inclusion into the segment. The name associated with the INCLUDE command is a library name and path sequence to the desired member of the library.

2. VERSION -- The VERSION command allows the user to specify a list of program library versions, determining the order of search through a program library for object modules when building segments.

3. SEGSIZE -- The SEGSIZE command limits the number of storage units which a segment may occupy. The predetermination of segment size has several uses. For instance, when segments are expected to be burned into ROM, the fixing of the segment size aids the user in the optimum organization and filling of the ROM space. Segment

sizing is useful when overlay operations are anticipated so that all overlays may be the same size and fit into a given area of memory. Overlays may contain several segments apiece, so the sizing of individual segments may prove to be useful for the construction of well fitting overlays. This command is optional.

4. SEGNAME -- The SEGNAME command associates a literal name with the bound segment. This command is generally be optional, but may be required when overlay operations are expected.

5. FILL -- The FILL command specifies a category of information with which to fill unused space at the end of a bound segment. The space may be filled with:

   a. Lexical descendants of included program units

   b. Externally referenced subprograms included internally

   c. Copies of external subprograms that have already been included in other segments.

6. OVERFLOW -- The OVERFLOW command allows the segment binder to form a second (or later) segment when the included contents of a segment exceed the specified or default segment size.

7. PARTIAL -- The PARTIAL command specifies that final resolution of references created by included program units during this binding session should not be accomplished. Once a reference has been called externally and set into the dictionary as "external" at the close of a binding session, it may not be changed to "internal". Partial binding allows the resolution of these references as "internal" in a later session.

8. EXPAND -- The EXPAND command allows the user to designate a bound segment to which further program units are to be bound. To ensure that the added program units are properly referenced, the segment should have been bound with the PARTIAL option in earlier sessions.

9. END -- The END command designates the end of the binding commands for a segment. The End-of-File mark actually designates the end of the binding operations, so it is possible to bind several segments during a single binding session.

The second source of input into the segment binder is the program units, named in the INCLUDE binder command. These units are retrieved from the program library as illustrated in Figure 3-2.

3.2.2.3.1.2 Processing

Segment binding is performed in two passes which allows a more organized approach to the construction of the run-time tables resulting in an optimally bound segment. The first pass interprets user commands and check the validity of program library references. The 'INCLUDE' command is the primary means of designating the contents of the segment. The algorithm describing the action of this command is as follows:

```
Pass 1:

Find compilation unit by library and path given in
     'INCLUDE' command
Check library directory section of dictionary for
    coded form of library name
If not found
    Add library name to directory
Endif
Check dictionary for library and path name
If found and is already bound
    Issue ERROR
    Exit Binder
Endif
If found and is external
    Remove from external list
    Decrement external count
Endif
Add entry to dictionary
Mark as internal
Enter library and path name
If object code has external references
    For each named external reference
       Search Ada program library for named
                           reference
       If found
          Find library and path in dictionary
          If not found
             Add entry to dictionary
             Mark as external
             Enter library and path name
          Endif
       Endif
    Endfor
Endif
```

Between the passes, the segment binder is responsible for performing the optimization of program units resulting from generic instantiations of the same generic definition. When generic optimization is requested, the library names of the effected program units are passed to the optimizer. The functional design of the generic optimizer is described in Section 3.2.1.6.10.3 of the Compiler CPDS. The returned optimized code is bound into the containing segment. The code section dictionary entries for the pre-optimized generic instantiations are not removed, but the displacement values for each are set to the new entry displacement. Other displacements in the table may have to be changed to allow for the packing of the segment by removing program units and the possible expansion of the newly optimized (by type casing) code.

The second pass brings together the compilation unit object code modules to form the bound segments. If the 'PARTIAL' binder command is specified for the binding session, this pass of the binder will not be executed. The 'PARTIAL' command allows the expansion of the section dictionary during multiple sessions. When the second pass of the segment binder is executed, external references are fixed, thereby setting the order of the members of the dictionary. This fixing does not allow further internal definition of formerly external references without a complete re-binding.

```
Pass 2:

For each internal reference in dictionary
    Find object code module in library
    Enter object code displacement to dictionary
    Increment segment size
    If object code has external references
       For each external reference
          Find library and path in dictionary
          For each reference along external
              reference chain
            Get reference to next link on chain
            Substitute dictionary entry index
                for chain link
          Endfor
       Endfor
    Endif
    Place object code module into section
Endfor
```

3.2.2.3.1.3 Outputs

The output from the segment binder is a file containing the bound segment which is inserted into the program library. The segment is now in a form that may be saved either by burning it into ROM for embedded processor deployment, or by storing it in a library file for debugging, testing or general usage. In either environment, the bound segment is ready to be combined with other bound segments to become a program. The parts making up the bound segment are as follows:

1. Pre-load Section Dictionary

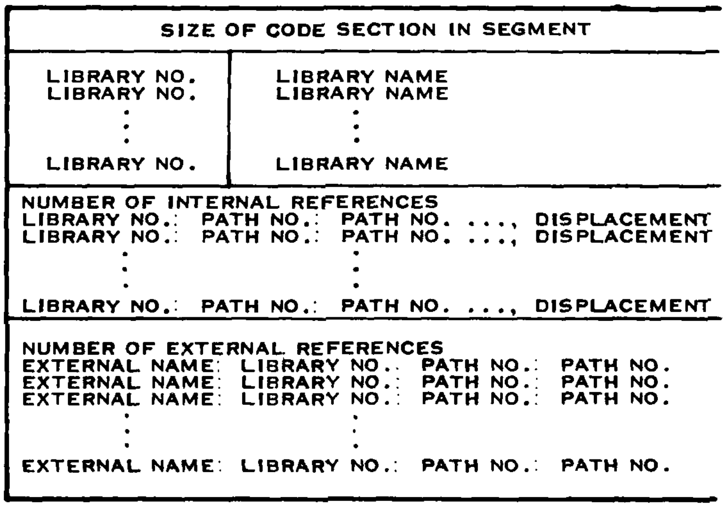
2. Loadable Code section

3. Constant Dictionary

4. Constant Section

3.2.2.3.1.4 Pre-Load Section Dictionary

The pre-load section dictionary is a block of information which informs the program binder of the contents and requirements of the loadable object section. The dictionary carries information dealing with the size of the code section, a directory of library names which are coded into the dictionary, a list of library memberships of internal subprograms so that references to this segment may be resolved, and a list of external references which need to be resolved by the program binder (external names are temporary). The layout of the pre-load section dictionary is shown in Figure 3-3.



SIZE OF CODE SECTION IN SEGMENT

LIBRARY NO. LIBRARY NAME
LIBRARY NO. LIBRARY NAME
⋮
LIBRARY NO. LIBRARY NAME

NUMBER OF INTERNAL REFERENCES
LIBRARY NO.: PATH NO.: PATH NO. ..., DISPLACEMENT
LIBRARY NO.: PATH NO.: PATH NO. ..., DISPLACEMENT
⋮
LIBRARY NO.: PATH NO.: PATH NO. ..., DISPLACEMENT

NUMBER OF EXTERNAL REFERENCES
EXTERNAL NAME: LIBRARY NO.. PATH NO.: PATH NO.
EXTERNAL NAME: LIBRARY NO.: PATH NO.: PATH NO.
EXTERNAL NAME: LIBRARY NO.: PATH NO.: PATH NO.
⋮
EXTERNAL NAME: LIBRARY NO.: PATH NO.: PATH NO.

Figure 3-3 Pre-Load Section Dictionary

3.2.2.3.1.5 Loadable Code Section

The loadable code section is a module of code which has been linked by the segment binder. This linkage may have been accomplished in multiple binding sessions. The completeness of the section is determined by the user. The user is also responsible for the specification of the program units making up the segment. All internal branches are resolved at this time.

Subprogram calls are resolved such that the object code need not be modified when program binding takes place. A unique method of determining subprogram entry addresses is maintained during runtime which is independent of the object code. The object code for a subprogram call consists of a subroutine branch to a subprogram call handler and a numeric value. The address of the call handler is available through an indirect reference to a dedicated register so that the branch to the call handler is handled externally to the segment. The numeric value is the entry index into the calling subprogram's section dictionary. Using the information found in the dictionary and other tables at its disposal, the calling utility is able to determine the address of the called routine. When the segment is introduced to the program binding phase, this information will be included in its pre-load dictionary. The system of tables will provide a transportable mechanism allowing independence from the resolution of external and internal references in the object code at final binding.

3.2.2.3.1.6 Constant Sections

The constant section and the constant dictionary are formed during segment binding. The result of that binding is a form which is ready for final program binding or loading. The constant section is a sequential grouping of the constants found within the program units of a segment. A constant is defined as a (non-scalar) constant that is allocated in code space (instead of data space) and is (1) passed by reference to another subprogram or (2) is large enough that it is not desirable to make a copy in each offspring subprogram where it is referenced in global scope. The constant dictionary is a structure which contains displacements from the beginning of the constant section for each program unit in the segment. These displacements mark the start of the constants as they are defined in each program unit. The entries in the constant dictionary match the internal entries in the section dictionary in a one-to-one correspondence.

3.2.2.3.2 Program Binding

The program binder is responsible for the formation of executable Ada programs. Figure 3-4 illustrates the overview of the processing which takes place during this phase of the AIE Binder.

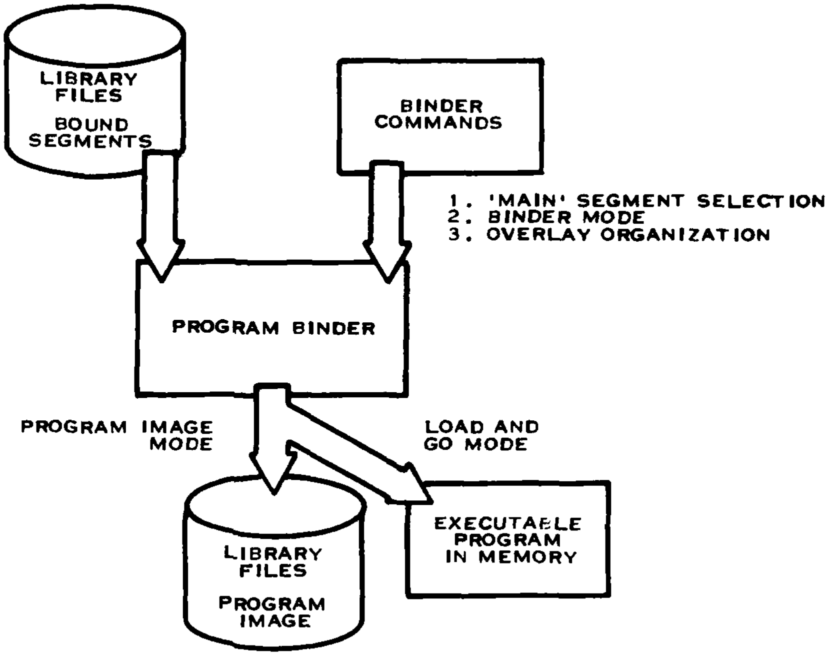


Figure 3-4 Program Binder Overview

3.2.2.3.2.1 Inputs

The program binder requires two sources of input for its proper operation. The user commands which determine the structure and composition of the final program. These commands are made available to the program binder through the KAPSE, but normally originate through a user terminal or control file in the library. The commands to the program binder are:

1. MAIN -- The MAIN command names the bound segment containing the program unit which is the root of the program.

2. OVERLAY -- The OVERLAY command lists the names of the bound segments which make up a program overlay. A numerical parameter to this command denotes the level of nesting of the overlay. That is, an overlay may include lower levels of overlaying, and overlays of the same numerical value will be paged into the same memory space. This command prevents the binder from using demand paging binding.

3. PROGSIZE -- The PROGSIZE command is an optional command which allows the user to limit the size of the memory space used by the bound program. This command aids the binder in the determination of the need for demand paging of segments when overlays are not specified.

4. STACKSIZE -- The STACKSIZE command is an optional command which allows the user to override the default or earlier specified stack size for the bound program.

5. HEAPSIZE -- The HEAPSIZE command is an optional command which allows the user to override the default or earlier specified heap size for the bound program.

6. IMAGE -- The IMAGE command specifies that the bound program image be placed into a named program image file in the program library. The program binder defaults to the load-and-go binder mode if this command is not in the command stream.

The second source of input into the program binder is bound segments generated by the segment binder phase. These units will be retrieved from the program library as illustrated in Figure 3-4.

3.2.2.3.2.2 Processing

The operational environment of the Ada program being bound dictates the mode(s) for the operation of the program binder. The program binder may operate in one of two modes with options attached to these.

1. Program Image Binding

2. Load-and-Go Binding

These modes may be augmented by further instructions to the binder to cause overlay operations or demand paging of segments.

3.2.2.3.2.3 Program Image Binding

The program image binder provides the user with a bound program image which is saved and used in multiple executions. The program image binder also provides debugger support for the quick and efficient development of Ada systems. The program image binder operates in two passes. The first pass is responsible for the determination of which segments will be needed for complete definition of the program. This determination starts with the examination of the external references contained in the pre-load section dictionary of a named main segment. External references will be added to a "Referenced and Resolved" list as each segment which resolves earlier external references is brought into the program. When all external references are resolved, a second pass will build the code sections of these segments into a program and will supply required information to the segment table and individual section dictionaries. This gathered information is made available to the user in the form of a subprogram binding map. This information will also be passed to the debugger for more thorough investigation of the operation of the program. The algorithms used in the program binder will be:

```
Pass 1:

Initialize segment table
Get 'Main' segment
Initialize program area
Build unresolved reference chain for first
    entry in main segment dictionary
For every unresolved entry in reference chain:
  Get segment table entry for segment
  Get segment info block for segment
  Enter segment library information into info block
  Find size of dictionary from pre-load dictionary
  Get runtime dictionary block of appropriate size
  Put address of dictionary block into segment table
  For all dictionary internal references
     Get next dictionary entry
     Mark as internal
     Enter code section displacement
     Find entry in reference chain
     If not found
        Build a reference chain entry
        Enter library nomenclature
     Endif
     Enter segment number
     Enter dictionary entry index
     Mark as resolved
  Endfor
  For all dictionary external references
     Get next dictionary entry
     Mark as external
     Find entry in reference chain
     If not found
        Build a reference chain entry
        Enter library nomenclature
        Mark as unresolved
     Endif
  Endfor
Endfor

Pass 2:

For all segments in segment table:
   Get segment info block
   Get segment from library
   Release segment info block
   Get segment's section dictionary
   Load code section into program area
   Load segment table with code section address
   Load constant dictionary into program area
   Enter constant dictionary address into segment table
   Load constant section into program area
   Enter constant section address into segment table
   For all external references in pre-load dictionary
```

```
            Find reference in reference chain
            Get segment number for dictionary entry
            Get dictionary index for dictionary entry
            Enter info into runtime section dictionary
         Endfor
         Find beginning of reference chain
         For all entries in reference chain for current segment
            Dump information to binder map
         Endfor
      Endfor .
      Release all members of reference chain
      Write program area, segment table and section
         dictionaries to program file
```

3.2.2.3.2.4 Load and Go Binding

The second mode of the program binder is basically a load-and-go binder. This load and go binder has the capability to perform the loading of segments (which under most circumstances will be residing in ROM) into an executable program with a minimum amount of memory, call a utility to set up necessary linkages, and execute the bound program. This binder may then be used during program development by providing an environment for the quick binding and testing of new programs.

The load-and-go binder also proves to be useful in the construction of programs from segments residing in ROM, such as may be found in an embedded processor system. Program binding of ROM segments allows the quick modification and customization of embedded systems with the interchange of ROM packages. In this way, the personality of a system may be altered significantly without re-binding all members of the system. Of course, with the bound segments in ROM, constructing the segment table and section dictionaries in RAM memory becomes the biggest responsibility of this binder. The load-and-go binder may be called during system power-up so that some crucial programs can be "loaded" and run quickly and efficiently.

The load-and-go loader constructs the executable program using two primary functions. The first deals with the establishment of the segment table for the segments involved in the program. All segments are assumed to be in ROM within certain address bounds. The second function builds the section dictionaries for the runtime environment. The code sections are re-entrant and do not have to be pulled from ROM into RAM memory. The algorithms which complete the load-and-go loading will be:

```
Establish Segment Table Entry:

   Starting address is to a pre-load dictionary
   Get size of code section from dictionary
   Get number of entries in dictionary
   Calculate starting address of code section from
      number of entries in dictionary
```

```
  Get next segment table entry
  Put code section address in segment table
  Put constant section address in segment table
  Put constant dictionary address in segment table


Establish Section Dictionary External Entry:

  Get starting address of segments in ROM
  Repeat
    Scan internal names of dictionary for external name
    If found
      Calculate starting address of code section from
        number of entries in dictionary
      Find segment table entry with correct segment
        address
      If not found
        Establish segment table entry for this segment
      Endif
      Save segment table index for section dictionary
      Save entry index in scanned dictionary for
          reference in current dictionary
    Else
      Calculate address of next dictionary from
        number of entries in scanned dictionary and
        code section size
    Endif
  Until external name is found or ROM bound is met
  If external name is found
    Add entry in runtime dictionary
    Mark entry as external
    Enter segment number
    Enter section dictionary index
  Endif

Binding Control:

  Initialize segment table and pointers
  Get starting address of main segment in ROM
  Establish segment table entry for main segment
  For every entry in segment table
     (segment table may grow dynamically in this loop)
    Find code section for segment
    Find section dictionary for segment
        (back track from code section)
    Find total number of entries for dictionary
    Get block for entries
    Put address of block into segment table for
       section dictionary
    For all internal members in pre-load.dictionary
       Get entry in runtime dictionary
       Mark as internal
       Enter code section displacement
      Endfor
      For all external members in pre-load dictionary
         Establish section dictionary external entry
      Endfor
    Endfor
```

3.2.2.3.2.5 Demand Paging Binding

LINK TO NEXT OCB
OVERLAY CONTROL FLAGS
DISK ADDRESS OF OVERLAY
STORAGE SIZE OF OVERLAY
CURRENT MEMORY ADDRESS
SWAP SCHEDULE INFORMATION

Figure 3-5 Demand Paging Control Block

Demand paging operations are initialized in the program binder and/or load-and-go loader. Pages are normally one or more segments which are superficially bound together in what the user deems to be a logical order. Bound pages will normally reside on a disk (when using the program binder) and are brought into memory when a program unit within the page is needed for execution. In the load-and-go binder, pages will normally be adjacent segments in ROM which mutually reside in a bounded address space. Paging operations will involve address switching to enable the proper ROM when a program unit in a given page is invoked. A system of tables and control blocks (Figure 3-6) are used to effect the demand paging operation. The paging director table is a list whose structure parallels the segment table. That is, the index into the segment table references paging information for the same segment. Entries for the paging director include a displacement into the page for the segment and a pointer to the demand paging control block for the containing page. During paging operations, the segment table entry for the code section address may be invalid because the address of the segment may vary from reference to reference as it is loaded into different areas of memory. The displacement in the paging director gives the call handler and other utilities needed information in establishing an entry address for an paged segment. The demand paging control block (Figure 3-5) is an information control center which:

1. Determines whether a page is in or out of memory

2. Points to the page's base address

3. Gives the page's size

4. Points to the page's disk/ROM image

Every segment which is included in any single page will have a pointer to a demand paging control block through the paging directory entry for that segment, with the exception of a segment which is to be memory resident at all times. When a segment is memory resident, the absence of a pointer to a demand paging control block will signal that the code section address in the segment table is valid.

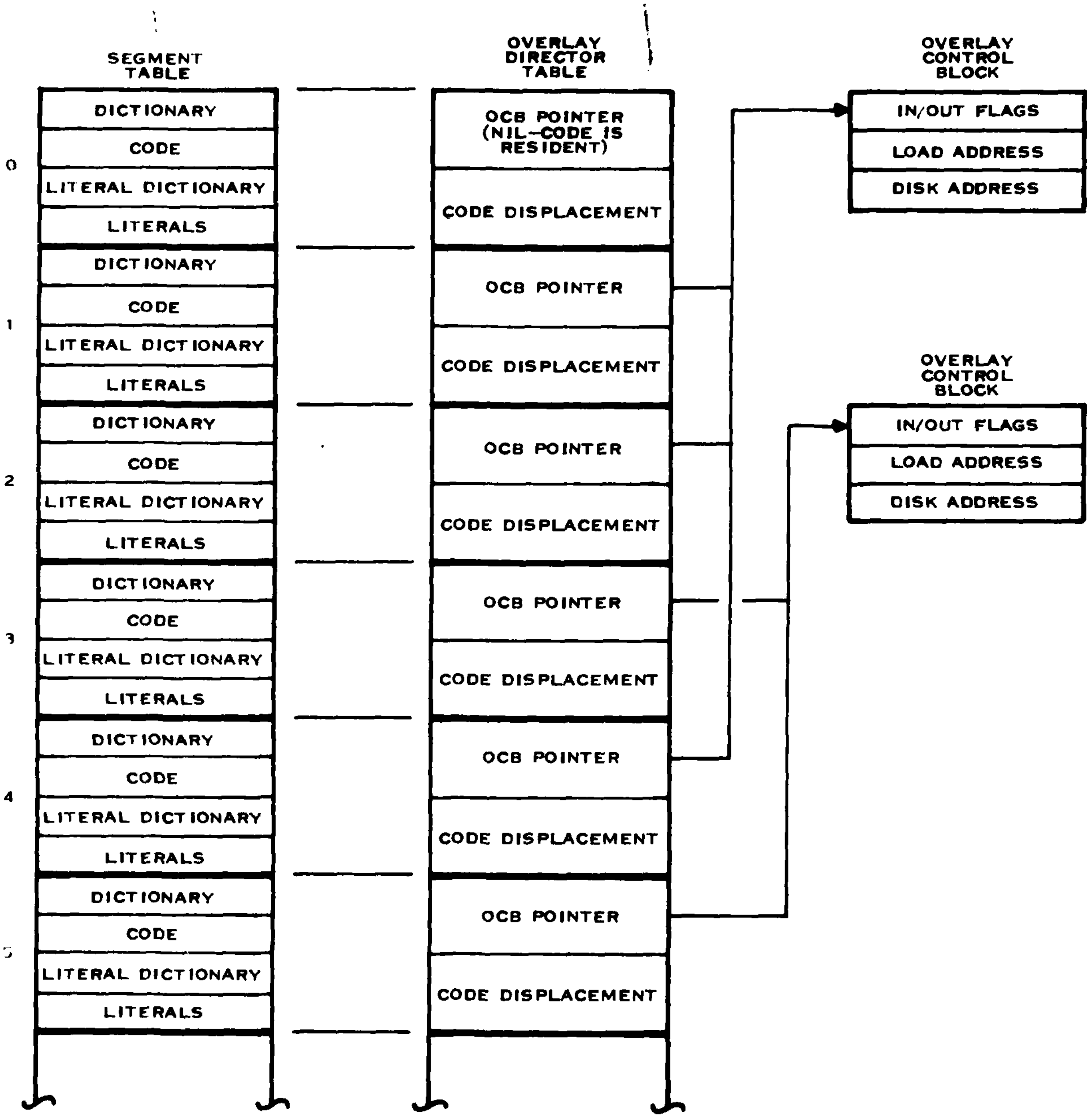


Figure 3-6 Demand Paging Control Structures

3.2.2.3.2.6 Overlay Binding

Another option to the program binder for the purpose of controlling the paging of segments and/or groups of segments is the specification of overlay operations. Overlays are special cases of demand paging. In demand paging, the bound segment groups are swapped into memory on a demand basis with memory allocation being a function of the available paging areas and scheduling algorithms. Overlays are bound segment groups which are paged into memory into pre-determined addresses. Flags in the demand paging control block specify that a bound segment group is an overlay. When this flag is set, the address found in the 'current memory address' entry of the demand paging control block is the fixed address of the overlay. The only entry in the block which will change is the in-out of memory flag. The KAPSE Interface Task (KIT) is responsible for the memory management of the overlay paging.

3.2.2.3.2.7 Outputs

The output from the program binder is a file containing the program image which is inserted into the program library when the binder operates in the Program Image mode. The binder will leave the program image in memory when the load-and-go binder mode is used. In either case, the environment, generated and output by the binder for the execution of the program, is virtually the same. The segments making up the program image consisted of four parts:

1. Pre-load Section Dictionary

2. Loadable Code section

3. Constant Dictionary

4. Constant Section

The program binder retains these four sections of the program segment. However, the program image will have transformed one of these parts by trading library pathnames for address displacements and table indices. The binder also generates other tables and blocks for the execution environment. These tables and blocks are:

1. The Segment Table -- The Segment Table manages the addresses of the four parts of all segments within a program.

2. The Global Package Table -- The Global Package Table manages the addresses of the storage areas used by packages which are global to the bound program.

3. The Program Parameter Descriptor -- The Program Parameter Descriptor describes to the program manager the expected parameters to the main subprogram of the bound program.

The following sections describe the output form of these segment parts when they are bound by the program binder.

3.2.2.3.2.8 The Segment Code Dictionaries

The code section dictionary goes through some drastic changes in the program binder. The most noticeable change is the transformation of the library name references into segment numbers as the external references are resolved. It is the transformed dictionary which is referenced from the segment table. As the binding progresses, external references are directed to loaded segments through the segment table. The resulting dictionary for the segment will change to the form in Figure 3-7.

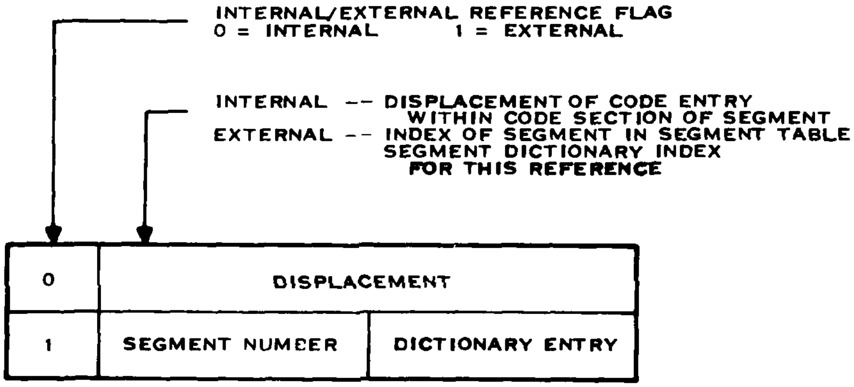

INTERNAL/EXTERNAL REFERENCE FLAG
0 = INTERNAL 1 = EXTERNAL
INTERNAL -- DISPLACEMENT OF CODE ENTRY WITHIN CODE SECTION OF SEGMENT
EXTERNAL -- INDEX OF SEGMENT IN SEGMENT TABLE SEGMENT DICTIONARY INDEX FOR THIS REFERENCE
0
DISPLACEMENT
1
SEGMENT NUMBER
DICTIONARY ENTRY

Figure 3-7 Segment Code Section Dictionary

The entries in the dictionary are flagged to differentiate the internal from the external references. The internal references (flagged with the "0") contain only the displacement of the subprogram from the start of the segment's object code. The address for the object segment is found in the segment table. External references (flagged with the "1") contain two pieces of information. The segment number identifies the segment in the segment table. With this information, the location of that segment's dictionary will be retrieved. The second piece of information within the original segment dictionary entry identifies the entry's index into the referenced dictionary. From this thread, the displacement and the address of the external reference will be discerned at run-time The address of the external reference will not be placed within the calling segment's dictionary as this may change during execution.

3.2.2.3.2.9 The Segment Table

The purpose of the segment table (Figure 3-8) is to keep an orderly list of the addresses needed in the run-time environment. Each entry to the segment table represents one segment in the program and contains four addresses. These addresses are:

1. Address of Code Section Dictionary
2. Address of Code Section
3. Address of Constant Section Dictionary
4. Address of Constant Section

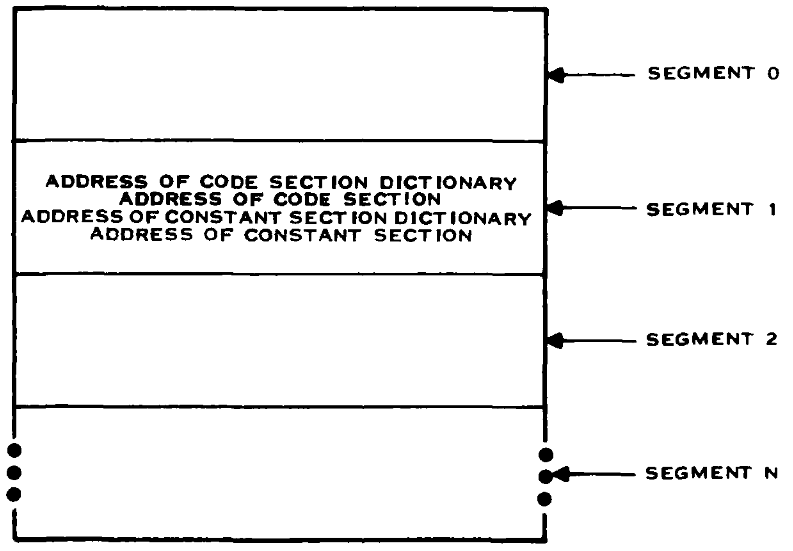

SEGMENT 0
ADDRESS OF CODE SECTION DICTIONARY
ADDRESS OF CODE SECTION
ADDRESS OF CONSTANT SECTION DICTIONARY
ADDRESS OF CONSTANT SECTION
SEGMENT 1
SEGMENT 2
SEGMENT N

Figure 3-8 Run-time Segment Table

Segments within the executable program are identified by their index into this segment table. Addresses within this table are entered as segments are brought into the binder and the run-time addresses calculated.

3.2.2.3.2.10 The Global Package Table

The global package table is constructed by the program binder whenever program units within the program segments contain an external reference to a global package. The global package table contains addresses to storage areas, allocated by the program manager, used by the package's visible variables. The reason for this table is to establish a means by which the executable code of the program, without the benefit of absolute addressing within the segment, will be able to access these visible parts of the package.

A global package handler is used to compute the address of a needed package's storage. The object code of the program unit is able to invoke this handler by an indirect branch to the address of the global package handler, kept in a system maintained table. This table also maintains the addresses of other pertinent handlers. A constant in the object code serves as an index into the global package table to determine the location of the visible parts of the package. This address is used as a base address for future references.

3.2.2.3.2.11 The Program Parameter Descriptor

The program parameter descriptor is a block of data describing the parameters to be passed to the main subprogram of the program so that proper execution of the program may occur. This block contains the number of expected parameters and the literal name, simple type and default value of each parameter.

The parameter descriptor is formed by the program binder from information about the parameters found in the symbol table for the program unit designated as "main". The information required for each parameter includes:

1. Name of Parameter -- The name of the parameter is derived from the identifier assigned by the user in the source code of the subprogram. The name should be descriptive of the actual usage of the parameter because the only modification of the name will be the elimination of underscores.

2. Simple Type of Parameter -- The simple type of the parameter is included in the descriptor so that some external type checking may be done previous to the parameter's usage in the program. The descriptor accepts the predefined simple types of INTEGER, BOOLEAN, STRING and CHARACTER.

3. Default Value -- The default value of the parameter, if available from the symbol table, is included in the descriptor. When the stack frame of the main subprogram is initialized, this value is used unless externally overridden with another value.

The program binder saves this program parameter descriptor so that the AIE Program Manager will be able to use it for invoking the program.

3.2.2.3.3 Special Requirements

The program binder is required to provide the execution environment with a complete program image. This environment has to provide a mechanism which allows for the efficient calling of subprograms and referencing of constants within the framework of the its data structures. The following sections describe the mechanisms used to effect the calling of subprograms and referencing of constants.

3.2.2.3.3.1 Subprogram Calls

It should be remembered that the segment's object code cannot be modified, but at the same time, the exact addresses of its called subprograms are unknown. To keep the calling sequences uniform, internal and external references are identical in the object code. The run-time environment is able to address a resident subprogram call handler. This utility is responsible for stack maintenance, parameter passing and the locating of the called subprogram. The finding of the called subprogram is the interesting function from the binder's point of view. The area in the object code which follows the branch to the call handler identifies the entry in the segment dictionary to which program control is to be passed. The subprogram call handler will use the following algorithm in finding the address of the called subprogram. Figure 3-9 gives a graphic description of the data structures and processing involved in a subprogram call.

```
Get segment number of current segment from
    Task Control Block
Get dictionary entry index from object code
Repeat
   Find segment entry in segment table
   Follow pointer to segment's dictionary
   Find referenced entry in dictionary
   If reference is 'EXTERNAL'
      Get segment number
      Get dictionary entry id
   Endif
Until 'INTERNAL' reference is found
Add displacement (from dictionary entry) to
   base address of segment (from segment table)
Result is entry address of called subprogram
```

An optimization in the addressing of calls to subprograms within the segment may be implemented. Since the postion of subprograms within the segment is known at segment binding time, the displacement of the called subprogram, relative to the code section, would be substituted for the section dictionary index. Knowing the entry address of the segment and the displacement, the called address would be calculated in a more efficient manner. This optimization requires that the section dictionary index word in the object code section be flagged according to its usage. A negative number in that position would indicate the dictionary index which would be complemented and applied.

The call handler algorithm is significantly changed when paging operations are being used. The following algorithm describes the call handler for paged segments.

```
Get segment number of current segment from
    Task Control Block
Get dictionary entry index from object code
Repeat
   Find segment entry in segment table
   Follow pointer to segment's dictionary
   Find referenced entry in dictionary
   If reference is 'EXTERNAL'
      Get segment number
      Get dictionary entry id
   Endif
Until 'INTERNAL' reference is found
Check demand paging operations flag
If paging is being used
   Get segment displacement from paging
      director table
   Get demand paging control block through
      paging director table
   If page is not in memory
      Swap in page from library/disk
      Enter load address into demand paging
         control block
      Mark page as in memory
   Endif
   Get page address from demand paging
      control block
   Add segment displacement for entry address
      for desired segment
Else
   Get base address of segment from segment
      table
Endif
Add displacement (from dictionary entry) to
   base address of segment
Result is entry address of called subprogram
```

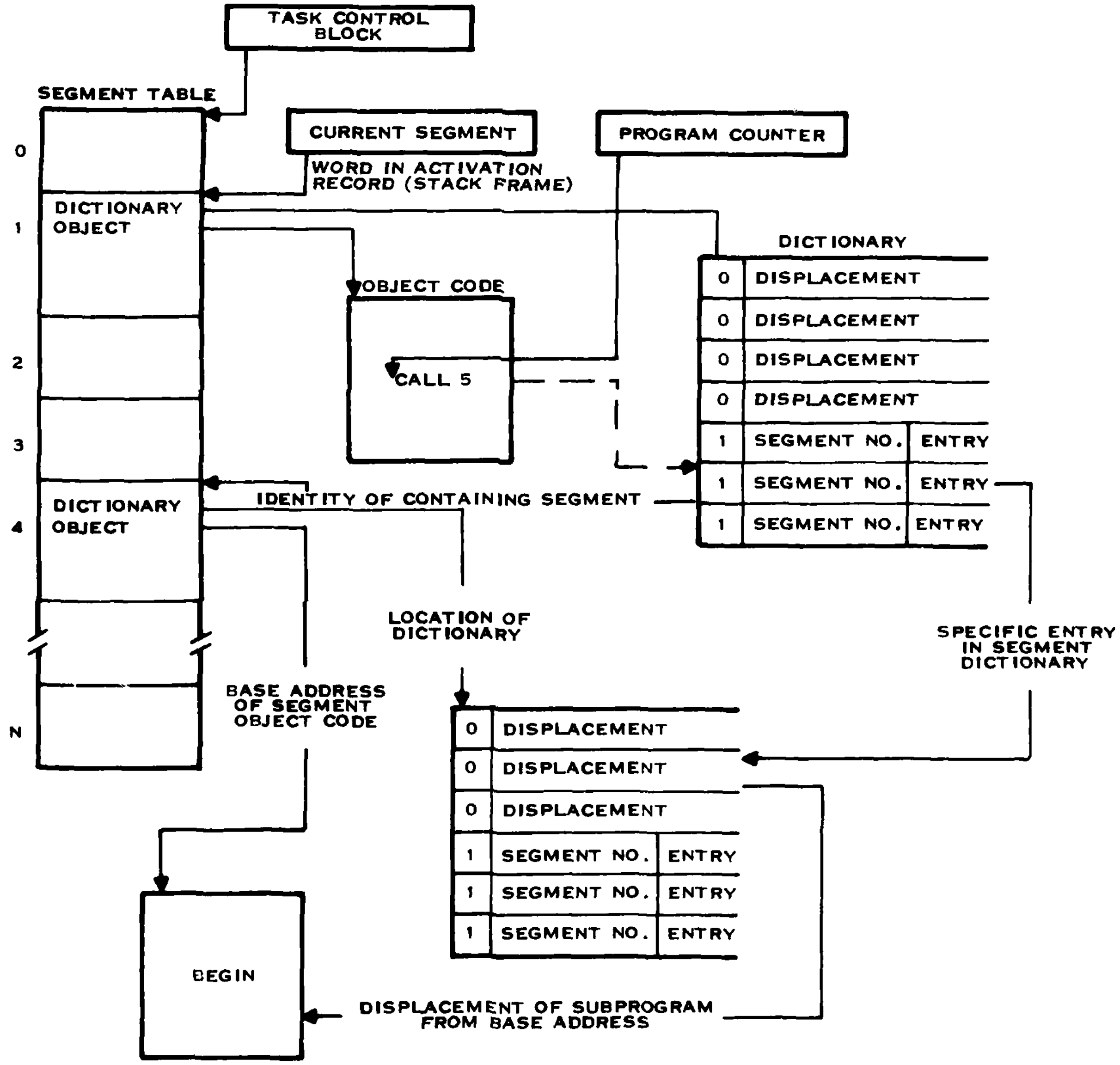


Figure 3-9 Subprogram Calling Linkages

3.2.2.3.3.2 References to Constants

The referencing of literal constants which are too large to be included in the code section is addressed in a manner similar to the calling of external subprograms. These constants are included in a constant section. An associated dictionary holds the displacements to the segment's program unit constant groups. These constructs were reviewed earlier. A constant handler is called as a part of the entry into a subprogram in which constants are used. Subsequent calls to the constant handler may be made when needed constants are out of the local scope of the subprogram. The code in the code section which calls the constant handler includes a storage unit of data to direct the handler to the referenced constant section. In a manner similar to the call handler, the constant handler's address is available to the run-time environment as a member in a block of addresses pointed to by a register or run-time variable. The attached storage unit is an index into the section and/or constant dictionary. There is a one-to-one correspondence between the entries in these two structures. The constant handler calculates the starting address of the constants for the segment section requested and deposits the address in a register or known variable. Requests for specific constants in that constant group are made by in-code offsets from the calculated address. The following algorithm describes the action of the constant handler in finding the address of a constant. Figure 3-10 and Figure 3-11 show the data structures and relationships which are used by the algorithm.

```
Get dictionary entry index from code
Get constant relative offset from code
Repeat
  Get section dictionary for segment
  Find dictionary entry for index
  If entry is external
    Get segment number
    Get dictionary entry index
  Endif
Until dictionary entry is internal
Get constant dictionary for segment
Get constant dictionary entry for index
Get displacement from entry
Get start of constant section from segment table
Address of constant group = Starting address of
    constant section + displacement
```

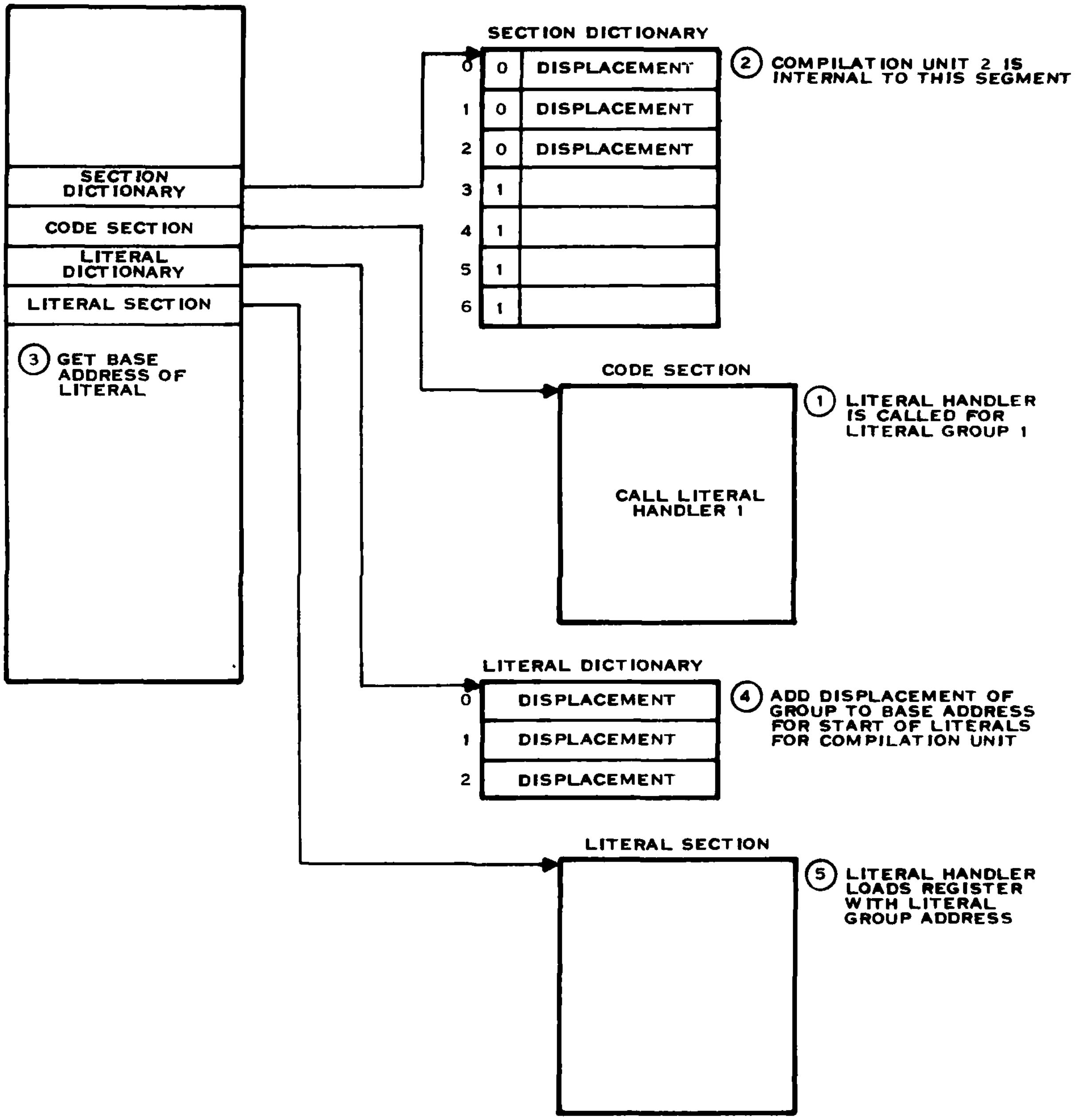

SECTION DICTIONARY
CODE SECTION
LITERAL DICTIONARY
LITERAL SECTION
3 GET BASE ADDRESS OF LITERAL
SECTION DICTIONARY
0 0 DISPLACEMENT
1 0 DISPLACEMENT
2 0 DISPLACEMENT
3 1
4 1
5 1
6 1
2 COMPILATION UNIT 2 IS INTERNAL TO THIS SEGMENT
CODE SECTION
CALL LITERAL HANDLER 1
1 LITERAL HANDLER IS CALLED FOR LITERAL GROUP 1
LITERAL DICTIONARY
0 DISPLACEMENT
1 DISPLACEMENT
2 DISPLACEMENT
4 ADD DISPLACEMENT OF GROUP TO BASE ADDRESS FOR START OF LITERALS FOR COMPILATION UNIT
LITERAL SECTION
5 LITERAL HANDLER LOADS REGISTER WITH LITERAL GROUP ADDRESS

Figure 3-10 Locating Address of Internal Constant Group

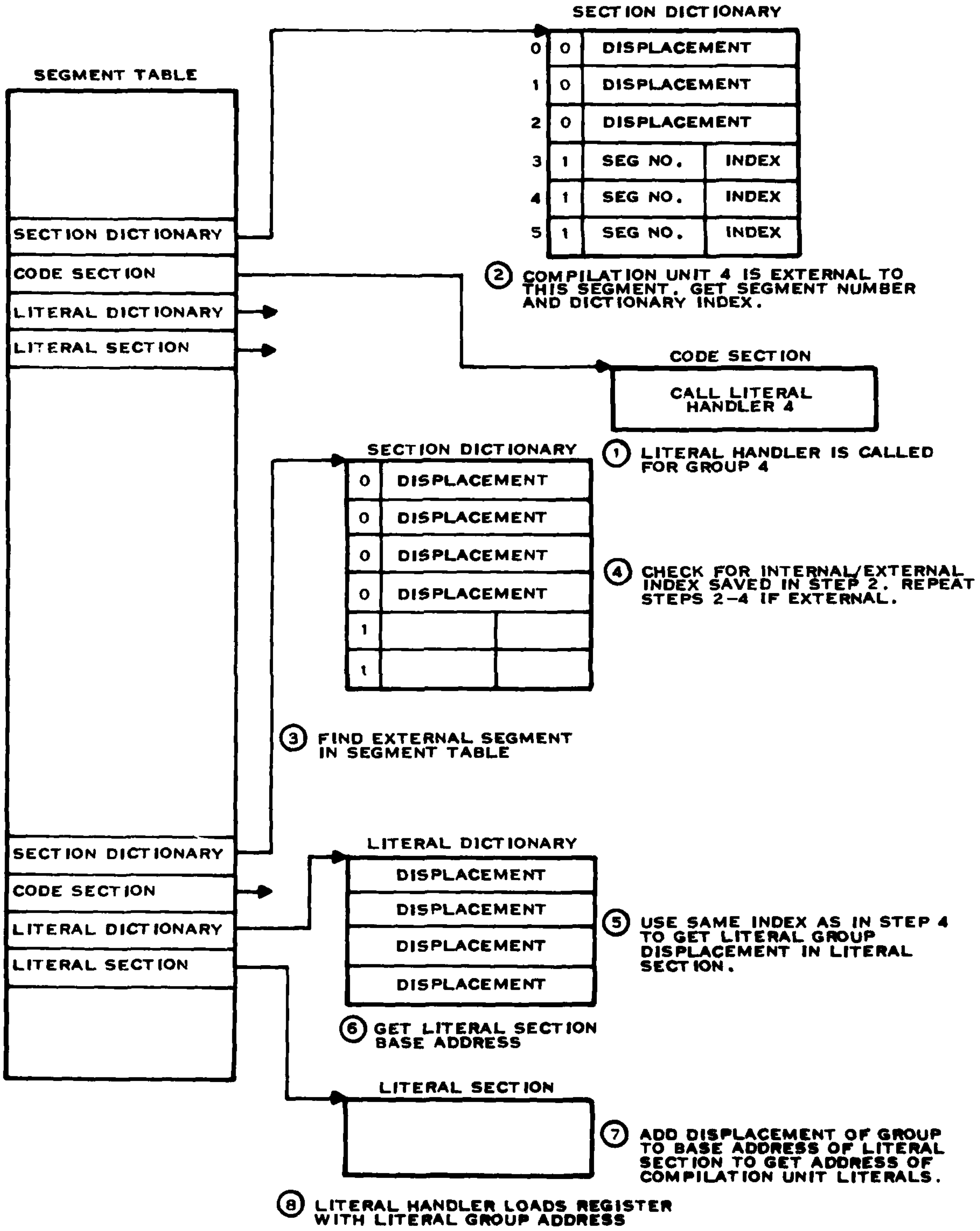

SECTION DICTIONARY
0 0 DISPLACEMENT
1 0 DISPLACEMENT
2 0 DISPLACEMENT
3 1 SEG NO. INDEX
4 1 SEG NO. INDEX
5 1 SEG NO. INDEX
SEGMENT TABLE
SECTION DICTIONARY
CODE SECTION
LITERAL DICTIONARY
LITERAL SECTION
② COMPILATION UNIT 4 IS EXTERNAL TO THIS SEGMENT. GET SEGMENT NUMBER AND DICTIONARY INDEX.
CODE SECTION
CALL LITERAL HANDLER 4
SECTION DICTIONARY
0 DISPLACEMENT
0 DISPLACEMENT
0 DISPLACEMENT
0 DISPLACEMENT
1
1
① LITERAL HANDLER IS CALLED FOR GROUP 4
④ CHECK FOR INTERNAL/EXTERNAL INDEX SAVED IN STEP 2. REPEAT STEPS 2-4 IF EXTERNAL.
③ FIND EXTERNAL SEGMENT IN SEGMENT TABLE
SECTION DICTIONARY
CODE SECTION
LITERAL DICTIONARY
LITERAL SECTION
LITERAL DICTIONARY
DISPLACEMENT
DISPLACEMENT
DISPLACEMENT
DISPLACEMENT
⑤ USE SAME INDEX AS IN STEP 4 TO GET LITERAL GROUP DISPLACEMENT IN LITERAL SECTION.
⑥ GET LITERAL SECTION BASE ADDRESS
LITERAL SECTION
⑦ ADD DISPLACEMENT OF GROUP TO BASE ADDRESS OF LITERAL SECTION TO GET ADDRESS OF COMPILATION UNIT LITERALS.
⑧ LITERAL HANDLER LOADS REGISTER WITH LITERAL GROUP ADDRESS

Figure 3-11 Locating Address of External Constant Group

3.2.3 Ada Interactive Debugger

The Ada Interactive Debugger provides the user with facilities to monitor a running Ada program and modify its target program flow, display and change the contents of its data structures, and display diagnostic information. The debugger is designed to provide these capabilities in interactive and non-interactive modes.

The design of the debugger permits the user to debug programs without any modification to the code generated by the Ada compiler. The debugger may be invoked at any point during the execution of the target program. The target program may be executing in the host machine or, if appropriate communications interfaces are provided, in some other target machine. The debugger may also be used as an interactive dump analyzer, when applied to a database object containing the image of the memory address space occupied by an Ada program.

The Ada Interactive Debugger consists of two major components. The Debugger Executive Program resides on the host computer and interfaces with the user, processes information, and accesses database objects. The Debugger Interface Package resides in the address space of the target program and provides the interface between the executive and the target program.

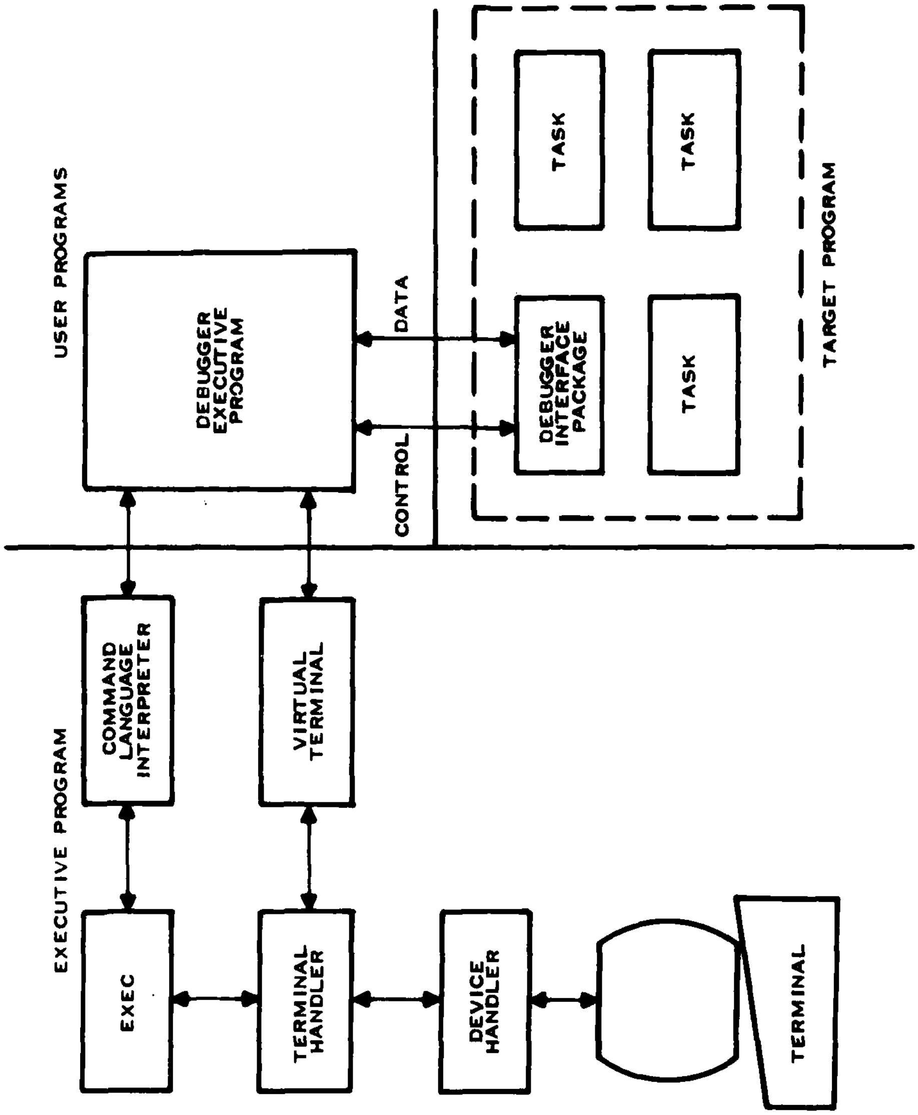


Figure 3-12 Debugger -- General Organization

3.2.3.1 Program Interfaces

The Ada Interactive Debugger is an Ada program, invoked by the user through the Command Language Interpreter (CLI). Once the Debugger is invoked, the target program is loaded (if necessary) in its own address space. Target program execution is suspended and control is passed to the debugger executive.

The function of obtaining information from and storing information into the address space of the target program is handled by the Debugger Interface Package (DIP), a component of the KAPSE Interface Task (KIT) which is contained in all invoked programs. Although the debugger interface runs as an independent task in the address space of the target program, it takes no action on its own but responds to commands from the Debugger Executive sent via inter-program communication facilities within the Kernel APSE. The processing of user requests, the collection and processing of information produced by the compiler and binder, the processing of files and the maintenance of debug internal variables are all under the control of the Debugger Executive.

The Ada compiler and program binder must generate several tables which may be required by the debugger, depending upon the features used. The debugger will be designed to be as machine-independent as possible; in addition, the system will be designed so that the debugger executive may reside on a host and the debugger interface on a target processor. Commands sent from the debugger executive to the debugger interface are at a high level to minimize the executive's need to know the target machine architecture. Since the debugger executive will be written entirely in Ada, the debugger may be rehosted on different architectures with minimum effort. All information passed to the debugger from the compiler and binder is contained in disk files related to the target program through the appropriate program library file.

The Ada Compiler must provide:

1. Statement Map

2. Symbol Map

3. Type Map

The Binder must provide:

1. Code Map

2. Ada Bound Program

The Kernel APSE must provide:

1. Program Location

2. Stack Pointers
3. Terminal I/O
4. File I/O
5. Software Interrupts which allow control to be passed to AID

3.2.3.1.1 Interface Block Diagram

Figure 3-13 shows the interfaces between the target program, the debugger interface, the debugger executive and the KAPSE. The input to the debugger interface, referred to as CONTROL, consists of commands generated by the debugger executive. Primary output from the debugger interface to the debugger executive will be values for display purposes, referred to as DATA.

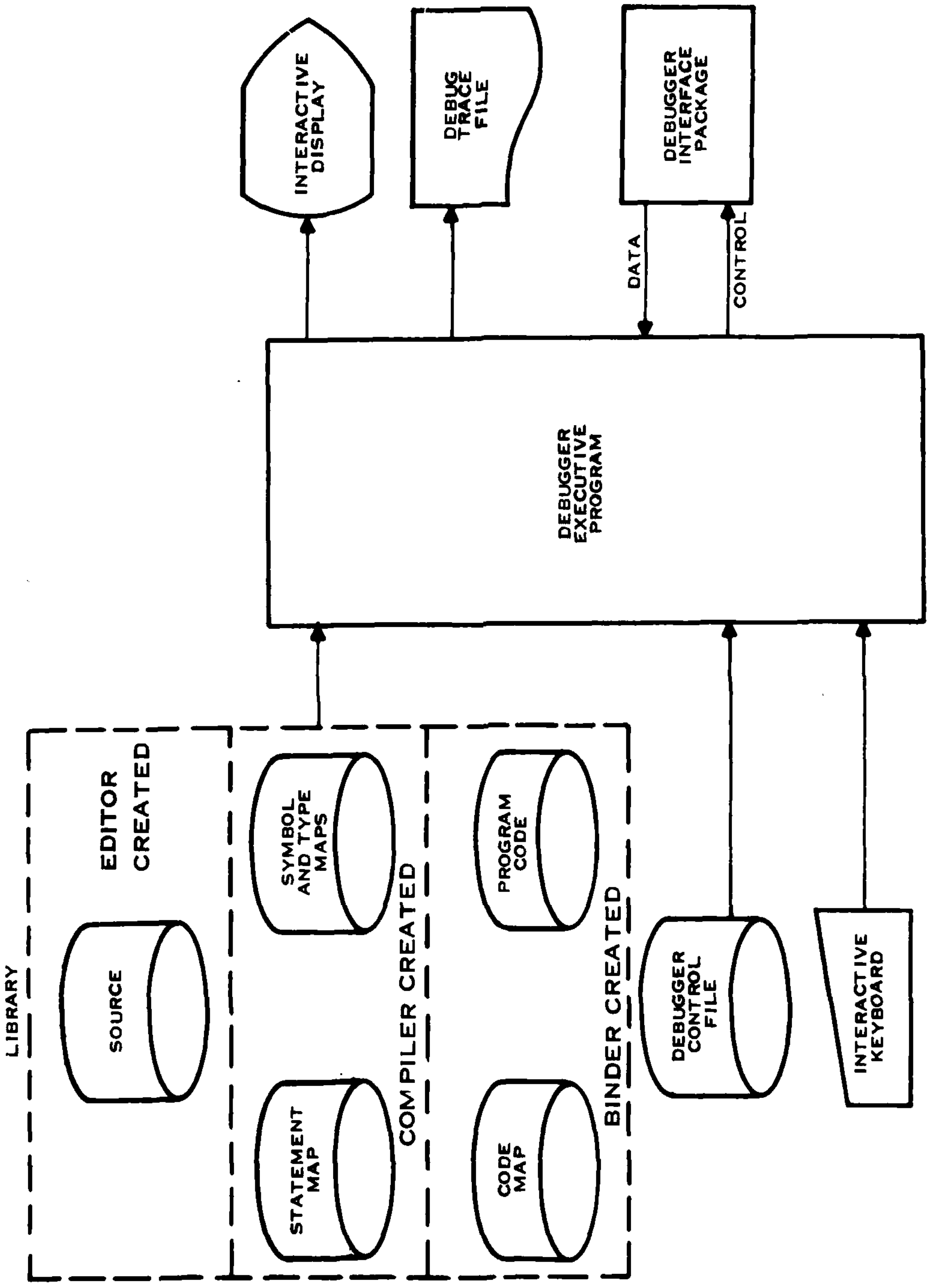


Figure 3-13 Debugger Interfaces

3.2.3.1.2 Detailed Interface Definition

This section describes the interfaces to be produced by the binder and the Ada compiler for the Ada Interactive Debugger. The information required will be generated automatically by the compiler and binder, and will be stored in files that are accessible to the debugger. Use of the Ada Interactive Debugger must not modify the way the binder and compiler perform their task; this is a major design criterion.

The information required to access symbolic entities within program units will be recorded in the symbol map and the type map. The symbol map contains a pointer to the corresponding entry in the type map. A statement map must be provided to facilitate mapping machine code back to the original source level statements.

The symbol map must contain the following information:

1. The name of the program unit or block.
2. The node_name of the symbol's entry in the symbol table.
3. The allocation of the symbol: register number, displacement in the code space, stack displacement or displacement in heap packet.
4. The name of the symbol's associated type in the type map.

The type map must provide the following information:

1. The name of the type within the type map.
2. The node_name of the type's entry in the symbol table.
3. The type descriptor.

The statement map must contain the following information:

1. The line number of the statement in the source listing
2. The source statement number.
3. The offset within the program unit or block of the first byte of the first machine instruction corresponding to the source statement.
4. The offset to the first byte of the last machine instruction of the source statement.
5. A pointer to the symbol table for each label on the statement.

The code map should contain the following information:

1. The name of each program unit within the program.
2. The length of each program unit and program segment.
3. The segment number of each program segment.
4. The ordinal position of each program unit within each segment.

The source listing must contain the statement numbers and a human-readable reproduction of the original source statement. This provides the user with the capability of displaying the original source text while using the debugger. The compiler will generate and display a name for each overloaded procedure on the listing so the user will have a unique identifier for each instantiation.

Two files are required from the binder: the bound Ada program ready for execution and a code map. The Ada Interactive Debugger places no additional demands on the object code. the purpose of the code map is to allow segment mapping within the bound Ada program back to the program units. In order to maintain the relationship between the statement map generated by the compiler and the bound Ada program, the binder should not modify the code within the compilation units.

The primary demands made by the Ada Interactive Debugger on other areas of the system are those placed on the executive and in particular the debugger interface. The debugger interface must have the these capabilities:

1. It must map the given segment number, entry number and displacement to a target machine address.
2. Given a CONTROL command to probe the code at a mapped address location, the debugger interface must return the original object code and replace it with the correct 'probing' code. A 'probing' code is object code which will interrupt execution of the target program, save the context of the target program and pass control to the debugger executive.
3. Given a target machine location as a segment number, an entry within that segment and an offset, it must start or resume execution of the target program at that location.
4. The debugger interface must inform the DEP when a task activation or termination occurs within the target program.

3.2.3.2 User Interface

The following paragraphs will describe the detailed input, processing and output functional requirements for the Ada Interactive Debugger.

3.2.3.2.1 Program Invocation

The Ada Interactive Debugger is designed to be used either interactively or in a batch mode. The debugger may be controlled with commands from a terminal or commands from a pre-existing file.

```
procedure DEBUG (NAME: in STRING; CONTROL: in STRING);
```

The debugger can be invoked with either one or two parameters. The required NAME parameter is the name of the program to be debugged or a dump file. The optional CONTROL parameter is either the name of a debug initialization file (a set of debug commands) or the keyword ACTIVE. If the second parameter is the keyword ACTIVE, the debugger assumes that the target program is already loaded and obtains information on its status from the KAPSE. The ACTIVE parameter puts the debugger in interactive mode. If the program is being run in batch, the two-parameter invocation is mandatory and the second parameter must specify the debug control file.

3.2.3.2.2 User Prompt

The Ada Interactive Debugger will display:

```
DEBUG:
```

when it is awaiting terminal input.

3.2.3.2.3 Command Language

The command language is Ada-like. The commands available to the user are:

```
statement ::=   abort_statement |
                accept_statement  |
                clear_statement  |
                define_statement |
                display_statement |
                dump_statement |
                end_terminal_statement |
                exception_statement |
                goto_statement |
                help_statement |
                if_statement |
                monitor_statement |
                run_statement |
                status_statement |
                symbols_statement |
                terminal_statement

sequence-of-statements ::= statement { statement }
```

3.2.3.2.3.1 ABORT Statement

```
abort_statement ::= ABORT ;
```

The ABORT statement is used to terminate a debug session.

3.2.3.2.3.2 ACCEPT Statement

```
accept_statement ::= ACCEPT breakpoint_name DO
                     sequence_of_statements
                     end {breakpoint_name};
breakpoint_name ::= selected_component | ALWAYS
selected_component ::= selected_name | statement_number
statement_number ::= integer
selected_name ::= identifier { . identifier } [ statement_number ]
```

The function of the ACCEPT statement is to enable the user to specify a set of commands which are to be executed when a given breakpoint occurs or upon the occurrence of every breakpoint. The ALWAYS parameter defines a set of debug statements that are to be executed at each breakpoint. ALWAYS could be used to trace the statements at which a variable changes values by indicating the new value.

```
ACCEPT ALWAYS DO
 DISPLAY( X, DB_Statement ); -- display information
END;                         -- end of global breakpoint
```

Example 3-1 ACCEPT Statement--ALWAYS

In Example 3-1, the ACCEPT command indicates that each time a breakpoint is executed, the debugger is to display the current value of the variable X and source text of the current statement.

The ACCEPT statement allows the user to give a set of commands which are to be executed when a breakpoint at a specific location is executed.

```
ACCEPT P.5 DO
  DISPLAY( X, Y, Z ); -- display variables
END P.5;              -- end of ACCEPT
```

Example 3-2 ACCEPT Statement--SPECIFIC LINE NUMBER

In Example 3-2 the ACCEPT statement indicates that if a breakpoint is encountered at statement number 5 in procedure P, the current values of the variables X,Y and Z are to be displayed. If a breakpoint is encountered and there is no corresponding ALWAYS or specific action, the results depend on the mode in which the debugger is being executed. In interactive mode, the breakpoint message is generated and control is passed to the user's terminal; in batch mode, the breakpoint message is generated and execution of the target program continues. A statement may be expressed either as a statement_number or as a label if there was a label on the original source statement.

3.2.3.2.3.3 ASSIGNMENT Statement

```
assignment_statement ::= name := expression ;
```

The debugger assignment statement may be used to assign new values to target program variables or to debugger variables. Assignment to constants is not allowed.

```
P.X := Q.Y + P.Z
```

Example 3-3 ASSIGNMENT Statement

In Example 3-3, the sum of the variable Y, defined in procedure Q, and variable Z, defined in procedure P, will be assigned to the variable X, declared in procedure P.

3.2.3.2.3.4 CLEAR Statement

```
clear_statement ::= CLEAR ( selected_component_list ) ;
```

The CLEAR statement is used to clear breakpoints which have been established via MONITOR statements.

```
CLEAR( P.X.5, Q.24 ); --clear breakpoints
```

Example 3-4 CLEAR Statement

In Example 3-4, the breakpoints at statement 5 in procedure X within procedure P and at statement 24 in procedure Q would be cleared. The execution of the CLEAR statement does not change the status of any ACCEPT statements which might be associated with the cleared breakpoints. The

ACCEPT statements are still maintained and will be executed, if a later MONITOR statement re-establishes the breakpoints and they are encountered during the course of execution.

3.2.3.2.3.5 DEFINE Statement

```
define_statement ::= DEFINE ( identifier,selected_component ) ;
```

The DEFINE statement allows the user to assign a selection string to a debug variable which will represent an entire selection prefix.

```
DEFINE( DB_P, P.L.Q.Z );    --set debugger string
```

Example 3-5 DEFINE Statement

In Example 3-5, the debugger variable DB_P is assigned the selection prefix P.L.Q.Z. When the user wishes to display the contents of the program variables N and X declared within procedure Z (within procedure Q within procedure L within procedure P), the user need only type DB_P followed by the variable name to select the proper variable.

3.2.3.2.3.6 DISPLAY Statement

```
display_statement ::= DISPLAY ( display_list ) ;
display_list ::= display_item { , display_item }
display_item ::= selected_component | display_function | literal
display_function ::= display_function_name ( identifier )
display_function_name ::= T | H | O | A | S
```

The DISPLAY statement is used to display the contents of target variables or debug variables during the course of the debug session. The display list is a list of variable names, line numbers, function invocations or literals separated by commas. The order in which the items appear in the list is the order in which they will be displayed. The format of the display output is the fully selected name of the variable, an equal sign, and the value, in the case of target program variables. If the variable is an array, the subscript will be displayed, enclosed in parentheses following the name of the variable in the output. The user may display all elements of an array by giving the array name. Specific items within the array may also be displayed by giving the desired subscript. If the variable is a debug variable, the name of the variable, an equal sign, and the value will be displayed. If the display_item is a statement number, the fully selected identifier for the source statement will be displayed on an output line followed by the source text on the next output line. The user may display information about variables or display values in a format other than the default. The options are:

1. T(variable)--Display the TYPE of the variable from the symbol table.

2. H(variable)--Display the value of the variable in hexadecimal.

3. O(variable)--Display the value of the variable in octal.

4. A(variable)--Display the address of the variable in the memory of the target processor.

5. S(variable)--Display the ranges of the subscripts of the variable.

```
DISPLAY( P.X, T(P.X), H(P.X) ); --display data on X
```

Example 3-6 DISPLAY Statement

In Example 3-6, the variable "X" defined within procedure P, would be displayed in the default format for its type. Then, the TYPE would be displayed and finally the value of the variable in hexadecimal would be displayed. Assuming the variable was REAL and had a value of 5.0, the display would be:

```
DEBUG:P.X=5.0
DEBUG:P.X=T(REAL)
DEBUG:P.X=H(41500000)
```

3.2.3.2.3.7 DUMP Statement

```
dump_statement ::= DUMP [ ( CODE ) ] ;
```

The DUMP statement gives a formatted dump of the stack, heap and, optionally, the code areas of the target program. The statement has at most one parameter. If the parameter is specified, the debugger will dump the code areas of the target program as well as the stack and heap. If the parameter is omitted, only the stack and heap will be dumped.

```
DUMP( CODE ); --dump stack, heap and code
```

Example 3-7 DUMP Statement

In Example 3-7, the debugger would give a formatted dump of the stack, heap and code areas. The dump of the stack and heap will be in historical order from the most recent stack frame to the oldest. The dump of the code segments will follow the dump of the stack and heap information and will be in the order in which the segments are produced by the binder.

3.2.3.2.3.8 END_TERMINAL Statement

```
end_terminal_statement ::= END_TERMINAL ;
```

The END_TERMINAL statement has meaning only when entered from a terminal following a TERMINAL command. The statement returns control from the terminal to the EXCEPTION or ACCEPT statement which originally passed control to the terminal. If control was passed to the terminal via a breakpoint without a specific action, this statement is identical to a RUN statement.

3.2.3.2.3.9 EXCEPTION Statement

```
exception_statement ::= EXCEPTION exception_handler  END ;
exception_handler ::= WHEN exception_choice
                       { | exception_choice } => exception_action
exception_choice ::= exception_name | OTHERS
exception_action ::= sequence_of_statements | SYSTEM
```

The EXCEPTION statement specifies a sequence of commands to be executed when an exception is raised within the target program. When the exception is raised, the exception choices are searched for the occurrence of the given exception name. If a match is found, the sequence of statements following that exception name is executed; otherwise, the sequence of statements following the keyword OTHERS is executed. If the keyword SYSTEM is used, it must be the only statement specified. This indicates that the normal system action is to be taken when the given exception is raised.

```
EXCEPTION
WHEN DIVISION_CHECK =>
  DISPLAY( "DIVISION BY ZERO AT", DB_Statement );
  DISPLAY( X, Y, Z, D );
  STATUS( TARGET );
  DUMP;
WHEN OTHERS=>
  SYSTEM;
  END;
```

Example 3-8 EXCEPTION Statement

In Example 3-8 when a DIVISION_CHECK exception arises the system will display the string "DIVISION BY ZERO AT", followed by the statement at which the error occurred. Next, the variables X,Y,Z and D in the current procedure will be displayed. Finally, the target program status will be displayed and the context of the target program will be dumped. If the exeception is not a DIVISION_CHECK, the normal system action will be taken.

3.2.3.2.3.10 GOTO Statement

```
goto_statement ::= GOTO ( name ) ;
```

The GOTO statement allows the transfer of control to a new source line within the current procedure of the target program. The name must be a label on a source statement. The user will not be allowed to transfer control outside the current block or procedure.

```
IF A.B = 0 THEN GOTO LX;
END IF;
```

Example 3-9 GOTO Statement

In Example 3-9, if the value of the variable B, declared in procedure A, is equal to zero, control will be passed to statement labeled LX in the current procedure. Note a run statement is not necessary when a jump is used to resume execution of the target program.

3.2.3.2.3.11 HELP Statement

```
help_statement ::= HELP ( name ) ;
```

The HELP statement enables the user to obtain:

- Syntatic and semantic information about debug commands
- Definitions of debugger variables and functions
- Definitions of exceptions
- Explanations of debug return codes

The following information is displayed by the various parameters :

- debug_statement-- syntatic and semantic information of the given debug statement

- * exception_name-- a description of the conditions which will cause the named exception to be raised
- * debug_variable_name-- a description of the value associated with the given debug variable
- * debug_return_code-- a description of the error which would cause the variable DB_RETURNCODE to be set to the given value
- * LIST-- a list of debug commands, variables, and functions

3.2.3.2.3.12 IF Statement

```
if_statement ::=  IF condition THEN
                     sequence_of_statements
                  END IF ;
condition ::= expression
```

The IF statement within the debugger can be used to test the status of a program or debugger variable and to execute a sequence of statements if the condition is satisfied.

```
IF DB_L5C MOD 10 = 0 THEN  -- is the counter divisible
  DISPLAY( X );
END IF;
```

Example 3-10 IF Statement

In Example 3-10, the debugger variable DB_L5C is used to count the number of times the breakpoint at statement 5 in procedure P is encountered. If the debug variable, DB_L5C, is divisible by 10, the value of the variable X is diplayed.

3.2.3.2.3.13 MONITOR Statement

```
monitor_statement ::= MONITOR ( selected_component_list ) ;
selected_component_list ::=
               selected_component { , selected_component }
```

The MONITOR statement is used to set breakpoints at specific statements, procedures or variables within the target program. Statement number 10 in procedure P, within procedure X, would be written as X.P.10. If the statement has a label, the label rather than the statement number may be used.

Statement number breakpoints occur at the first machine instruction corresponding to the source statement. Procedure breakpoints occur at the entry and exit points of the procedure. Variable breakpoints occur when the variable's value changes.

```
MONITOR( X.P.10, Z.5, Q.P.7 );  --set breakpoints
```

Example 3-11 MONITOR Statement--Statement BREAKPOINT

In Example 3-11, the MONITOR statement causes breakpoints to be established at statement number 10 in procedure P within procedure X, at statement number 5 in procedure Z and at statement number 5 in procedure P within procedure Q.

```
MONITOR( X.MAN, Q.Z.N, P.STR ); --set breakpoints
```

Example 3-12 MONITOR Statement--VARIABLE BREAKPOINT

In Example 3-12, a breakpoint would occur when the following variables change: variable MAN declared in procedure X; variable N declared in procedure Z which is in procedure Q; variable STR declared in procedure P. Variables in the list must be variables in the target program and not debugger variables.

```
MONITOR( X.P, Z, Q.L );  --set breakpoints
```

Example 3-13 MONITOR Statement--PROCEDURE BREAKPOINT

In Example 3-13, breakpoints would be set at the entry and exit points of procedure P within procedure X, procedure Z, and procedure L within procedure Q.

3.2.3.2.3.14 RUN Statement

```
run_statement ::= RUN ;
```

The RUN statement causes the debugger to start execution of the target program or to resume execution following a breakpoint or exception.

3.2.3.2.3.15 STATUS Statement

```
status_statement ::= STATUS ( name ) ;
```

The STATUS statement is a machine-dependent instruction which, when executed, dumps one of the following: the status of the target program, the current status of the hardware of the target machine, the status of the system, the "set" statement breakpoints, the "set" procedure breakpoint or the "set" variable breakpoints. If the keyword SYSTEM is specified as the parameter, the available information on the status of the target hardware is displayed. If the keyword TARGET is given, the status of the target program is displayed. The parameter S will produce a list of the "set" statement breakpoints; V will produce a list of the "set" variable breakpoints; and P will produce a list of the "set" procedure breakpoints. For some options, the status information depends on the architecture of the target computer; however, typical data could be register contents, condition codes, or program counter contents.

3.2.3.2.3.16 SYMBOLS Statement

```
symbols_statement ::= SYMBOLS ( selected_component ) ;
```

The SYMBOLS statement allows the user, to display all symbols and their associated types declared within a given procedure.

```
SYMBOLS( P.X ); --display symbols and types
```

Example 3-14 SYMBOLS Statement

In Example 3-14, all symbols appearing in the symbol table for the procedure X within procedure P and their associated types will be displayed.

3.2.3.2.3.17 TERMINAL Statement

```
terminal_statement ::= TERMINAL ;
```

The TERMINAL statement is used to return control to the user's terminal when the debugger is used in the interactive mode. The terminal user may enter any debugger commands he wishes once he has control. This includes the modification of EXCEPTION or ACCEPT statements. The user returns control to the debugger by issuing an END TERMINAL or RUN command.

3.2.3.2.4 Internal Variables

Within the Ada Interactive Debugger there are two classes of internal variables. The first class lets the user calculate and store values for later calculations or comparisons. The second class is built into the debugger and is used to initiate features or to obtain values.

The internal variables are differentiated from program variables by their first characters,which must be "DB_". Since the user may need additional variables during the course of his debugging session, internal variables of the first class will not be explicitly typed. The TYPE of the variable is defined by its initial assignment. If a later assignment attempts to redefine the TYPE of a variable, an error message will be generated and the assignment statement will be ignored. All symbol table entries and storage necessary to support internal variables will be maintained by the Ada Interactive Debugger on the host machine. TYPE is limited to BOOLEAN, CHAR, INTEGER and REAL.

Variables of the second class have predefined TYPE and are created by the invocation of the Ada Interactive Debugger. The variables of this class are identified as either read-only or read-write. The only difference between the two is that read-only variables may not be the target of assignment statements. Read-only variables are, in general, those which return information on the run-time environment of the program being debugged. Read-write variables are, in general, those which invoke features of the debugger. If more than one task is active, these variables will refer to the task in which the breakpoint occurred. The read-only variables are:

* DB_LEVEL--Contains the level of recursion of the current routine in the current context.

* DB TASK--Contains the identification of the current task.

* DB_STACK--Contains a character string containing the description of the current active stack frames.

* DB_Statement--Contains the identification of the current statement.

* DB_SStatement--Contains a list of the currently set statement breakpoints.

* DB_PROCEDURE--Contains the name of the current procedure.

* DB_SPROCEDURE--Contains a list of the currently set procedure breakpoints.

* DB_VARIABLE--When a variable breakpoint occurs, this contains the name of the variable causing the breakpoint. It is null, if the breakpoint was caused by a break other than a variable breakpoint.

* DB_SVARIABLE--Contains a list of the currently set variable

breakpoints.

- DB_RETURNCODE--If the previous debug statement was executed successfully, this variable will be set to zero. Otherwise, it will be set to an integer number indicating type of error.
- DB_DATE--Contains the current julian date.
- DB_TIME--Contains the current time.

The read-write variables are:

- DB_SINGLE--This is a boolean variable which if set to "true" puts the debugger in single step mode. If it is "false", the debugger runs in multi-statement mode. The initial value of this variable is "false".
- DB_ECHO--This is a boolean variable which, if set to "true", causes the debugger to produce a file suitable for printing containing the debug commands and their output. The user may change the value of the variable during the course of a debug session with no change to any information already generated. The initial value of this variable is "false".
- DB_TRACE--This is a boolean variable which if set to "true" causes the debugger to generate statement track information. The track display is in the form of a statement_specification. The initial value of this variable is "false".

3.2.3.2.5 Internal Functions

Within the debugger, there are internal functions which allow the user to obtain information about the target program or target program variables. These functions are identified by the prefix "DBF_".

3.2.3.2.5.1 DBF_ATTRIBUTE

```
function DBF_ATTRIBUTE(X,Y:STRING) return STRING;
```

The DBF_ATTRIBUTE function has as its first parameter a STRING which is the selected name of a target program variable. The second parameter is a STRING representing an entry in the type map generated by the compiler. The function returns a string which is the entry in the type map for the given variable and attribute. If the attribute is not stored as a STRING in the type map, it will be converted to a STRING by the function.

3.2.3.2.5.2 DBF_DEFINE

function DBF.DEFINED(X : STRING) return STRING;

The DBF DEFINED function has a variable name as its parameter (this need not be a selected name) and returns a string which is the selected name of the procedure in which the variable is defined. If the variable is not defined within the current scope, a string of zero length will be returned.

3.2.3.2.5.3 DBF_TASK

function DBF_TASK return INTEGER;

The function DBF_TASK returns the number of tasks activated by the target program.

3.2.3.2.5.4 DBF_TASK_STATUS

function DBF_TASK_STATUS(X:STRING) return INTEGER;

The parameter passed to DBF_TASK_STATUS is a string which specifies a task within the current target program. The function returns an integer number indicating the status of the task.

3.2.3.3 Processing Requirements

3.2.3.3.1 Symbol Identification and Completion

Symbols are entities declared within the target program. The AID accesses them using the symbol table and type table generated by the compiler. The symbol must be entered exactly as it appears within the program. A symbol entered without a selection criterion is assumed to be declared within the currently executing program unit. To reference symbols in other program units, the user must write the symbol as a selected component. If the symbol does not exist, an error message will be sent to the terminal in interactive mode or to the debug output file in batch mode. The statement containing the invalid symbol reference will not be executed; but, the syntax scan will continue. The user may use the SYMBOLS command to obtain a list of all symbols declared within a procedure.

Symbols are resolved in the following manner:

1. If the name starts with DB_, the debugger's internal symbol table is used.

2. If the name is a program name ,

   a. The symbol table is scanned for the symbol. If the symbol is not found, an error message is generated and scanning terminates.

b. If the symbol is found then the pointer to the corresponding entry in the type map is used to locate the correct entry in the type map.

c. The type is validated to test for any type conflicts.

3.2.3.3.2 Statement Identification

The Debugger identifies source text statements via the corresponding source statement numbers. These numbers will reflect those which appear on the compiler listing. A statement number entered without selection information is assumed to be within the currently executing program unit of the target program. To reference statements in other program units, the user must write the statement number as a selected component in which the last specification is an integer. The statement map generated by the compiler will be used to verify the statement number. If the statement map does not contain the statement, an error message will be generated, and the debug statement containing the invalid reference will not be executed. An invalid statement number does not stop the syntax scan.

3.2.3.3.3 Expression Evaluation

```
expression ::= relation { logical_operator relation }
relation ::= simple_expression
             [ relational_operator simple_expression ]
simple_expression ::= { unary_operator } term
             { adding_operator term  }
term ::= factor { multiplying_operator factor }
factor ::= primary [ ** primary ]
primary ::= literal |  name | ( expression )
logical_operator ::= AND | OR | XOR | AND THEN | OR ELSE
relational_operator  ::=   =  |  /=  |  <  |   <=  |  >  |  >=
adding_operator ::= + | - | &
unary_operator ::=  + | - | NOT
multiplying_operator ::=  * | / | MOD | REM
numeric_literal ::= decimal_number
decimal_number ::= integer [ . integer ]
integer ::= digit { [ underscore ] digit }
character_string ::= " { character } "
name ::= identifier
literal ::= numeric_literal |  character_string
          | NULL | TRUE | FALSE
```

The syntax of expressions used by the AID is similar to the syntax of expressions within the Ada language. Numeric symbols are considered constants of the type defined by their representation. Numeric symbols, debug variables and target program variables may be combined with mathematical or logical operators to form expressions. The AID supports all operators given in section 4.5 of the Ada Reference Manual and the precedence of the operators is identical to that of the Ada language.

In expression evaluation, the type rulés of the language apply. Symbols of unlike types may not be mixed in the expression. If an expression contains mixed types, the statement containing the expression will not be executed but syntax checking will continue.

3.2.3.3.4 Keyboard Interrupts

During program execution, the user may disconnect a program from a terminal using the ATTENTION key. This allows the user to halt the program via executive commands and invoke the debugger in an interactive mode. Since the ATTENTION key disconnects the target program from its window of the physical screen, the user must re-establish the connection between the target program and the physical terminal prior to issuing a RUN command from within the AID. If the RUN command is issued without ŕe-establishment of the link between the target program and the physical terminal, control will be given to the target program but terminal I/O will be suppressed.

3.2.3.3.5 Debugger Activation

Invocation of the debugger causes loading and initiation of the debugger executive. Prior to passing control to the target program, the debugger executive must do the following:

* Process the parameters. This particularly involves checking the second parameter:

  - If the ACTIVE parameter is not specified, the debugger executive must request that the KAPSE load the program. Otherwise, the debugger executive must obtain information on the status of the program from the KAPSE.

  - If a control file is requested, the debugger executive must read it in and process the control file.

* Obtain the binder's code map for the target program and translate it into internal format.

* Give control to the user's virtual terminal.

3.2.3.3.5.1 Parameter Processing

The CLI supplies the parameters to the debugger executive. The first parameter must be be the name of the Ada program to be debugged. The second parameter is optional; if present it may be either the name of a con file or an indication that the program is already active. Should the sec parameter be omitted, a check must be made to insure that the debugger is interacitve mode; otherwise, execution terminates with an error message.

3.2.3.3.5.2 Obtain Code Map

Using the name of the target program (parameter one) and the pro attributes, a request is made to the Data Base Manager (DBM) that access granted to the code map produced by the binder. If access is not grant an appropriate error message is generated and the debugger terminates. access is granted, the file is read and translated from human readable fo into an internal format. The internal format must allow efficie processing when mapping from a program unit name to an internal location the reverse.

3.2.3.3.5.3 Load or Link to Target Program

If the second parameter is not the keyword ACTIVE, the debugger executi requests that the KAPSE load the program and pass control back when t loading is completed. Should the program already exist, a message is se via Intertask Communication Facilities (ICF) to establish a link between t debugger executive and the debugger interface. If the program must loaded, the link is established after control is returned to the DE following the load.

3.2.3.3.5.4 Process Control File

If the second parameter is present and is not the keyword ACTIVE, t parameter designates the name of a control file and a request is made to t Database for access to it. Should access be denied, the DEP generates error message. In interactive mode, control is passed to the user's virtu terminal; in batch mode, the debugger terminates.

At this point, the debugger executive reads the control file sequentiall All statements are scanned for syntax and, if in error, a message generated. Statements within EXCEPTION or ACCEPT statements are scanned f syntax but not executed. If the statement is an EXCEPTION or ACCEP statement at the top level, the statements are processed and written to a internal debug direct access file keyed on the associated statement numbe No entry is made in the file for EXCEPTION or ACCEPT statements neste within other EXCEPTION or ACCEPT statements until the parent statement actually executed. All debug command processing is interpretive; therefor data written to the temporary file is in human-readable format.

MONITOR statements requesting statement level breakpoints require that th debugger have access to the statement map generated by the compiler. If request is made to set a breakpoint, debugger executive will determin whether the statement map for the given program unit has already bee processed. If so, the statement number is entered in a pending list and flagged as being a direct request for a statement breakpoint. Otherwise, a request is made to the DBM for the given program unit's statement map. If the access is successful, the map is read and processed. Should an error occur, a message is generated and a flag is set indicating to the debugger executive that statement level debugging will not be permitted in that program unit.

The request for variable breakpointing requires not only the statement map, but also the type map and the symbol map. A variable breakpoint requires that the system process statement maps for all program units as well as a symbol map and type map for the program unit in which the variable is declared. Statement maps are processed as described above. If the statement map is not available for any program unit, an error message is generated and the variable breakpoint occurs at the procedure level rather than the statement level. All statements in all program units are added to the pending list. Symbol maps and type maps, like statement maps, are initially requested from the DBM and then the debugger executive maintains them internally from that time on. A symbol breakpoint error message is generated under the following conditions:

- The type map for the given program unit does not exist.
- The symbol map for the given program unit does not exist.
- The symbol is not on the symbol table for the program unit.

In any of these cases, the statement is ignored.

Procedure level breakpoints do not require any file access to the Database since the information necessary to locate them is contained in the code map generated by the binder which is processed at the start of the debug session. The procedure level breakpoint is added to the pending list and identified as a breakpoint set by a direct request.

All breakpoint information both statement breakpoints or procedure breakpoints is maintained in a table containing the following:

- Breakpoint Identification
- The segment number, entry number, and displacement value of the first byte of the first machine instruction of the breakpoint.
- The original code contained in the above byte.
- The segment number, entry number, and displacement value of the first byte of the last machine instruction of the breakpoint.
- The original code contained in the above byte.
- A pointer to any ACCEPT/EXCEPTION table, if there is an entry for this statement.
- A flag indicating if this breakpoint was explicitly "set".

A statement breakpoint or procedure breakpoint in a MONITOR statement will generate an entry in the breakpoint table for the specified statement or procedure. A variable breakpoint in a MONITOR statement or entry into single step mode will generate an entry in the breakpoint table for every source statement in the program.

Once a MONITOR statement has been processed, then and only then are the pending breakpoint lists processed. Processing breakpoints requires requesting the DIP sent the first byte of the first and last instruction of the statement to the executive. This is then entered in the breakpoint table. Once this is accomplished, the DIP is told to place "set" interrupts in the first byte of the first instruction of the statement. If an error in syntax occurs during the processing of the statement, the pending lists are deleted and no breakpoints are set.

A RUN statement outside of an EXCEPTION or ACCEPT statement causes the debugger executive to stop reading the command file and pass control to the target program. In interactive mode, since there is no way to read the rest of the file once the debugger executive has passed control to the target program, a message will be written stating that any other commands on the command file will be ignored. In batch mode, the system will start reading commands from the command file following a RUN statement if:

1. A breakpoint occurs at a statement where there is no action specified

2. A procedure breakpoint occurs and there is no "ALWAYS" statement

3. A variable breakpoint occurs and there is no ACCEPT statement active

NOTE: If the debugger is in single step mode, it behaves as if there is a statement level breakpoint "set" at every statement in the program.

If an end-of-file is encountered in the control file prior to a RUN statement, results again depend upon the mode; in interactive mode, control is returned to the user's virtual terminal, while in batch mode, a RUN statement is assumed at the end-of-file. The first time the end of file is encountered in batch mode control is passed to the target program. Any additional attempt to read from the control file will result in an error message and termination of the debug session. An ABORT statement will terminate the debugger at once with all remaining information on the controi file ignored. It is suggested that the last command on a control file be an ABORT command.

3.2.3.3.6 Input/Output

All input/output to the terminal and debug output files is done by the debugger executive via Ada high-level I/O. The information transmitted, whether input or output, is in strings. Formatting and decoding is done by the debugger executive. Files within the DBM are accessed using standard access methods defined within the Ada language.

3.2.3.3.6.1 Virtual terminal I/O

The attachment of the user's virtual terminal to the debugger executive shall be the responsibility of the KAPSE. The internal file name will be TERMINAL. All terminal I/O is handled one line at a time. No attempt will be made within the debugger executive to do full screen formatting.

3.2.3.3.6.2 Debug file I/O

The debugger executive shall request the DBM to create a new entry in the directory under the target program named DEBUG. This file will have attributes of TIME and DATE which are the time and date at which the debugger is invoked. The user uses these attributes to differentiate between different debug files for the same target program.

3.2.3.3.7 Statement Processing

All statements in the debugger are executed via an interpreter within the debugger executive. Debug statements can be broken into two areas: those that are executed when encountered and those that are executed only when they are triggered by an event in the target processor. EXCEPTION and ACCEPT statements are the only statements which fall into the second group; all other statements are executed when encountered. Statements contained within EXCEPTION and ACCEPT statements are scanned for syntax as they are input but their execution is delayed until the EXCEPTION or ACCEPT statement is executed.

3.2.3.3.7.1 EXCEPTION Statement Execution

As exception statements are encountered, they are scanned for appropriate syntax and written to a relative record direct access file. An entry is made in a debug table giving the name of the exception, the number of statements and the location on the file. If the statement specifies the SYSTEM option, the exception is removed from the table and the space on the direct access file is marked as "available".

When an exception is raised in the target program, any exception processing indicated within the target program is performed and the debugger interface passes control to the debugger executive. The debugger executive searches the table to locate a corresponding EXCEPTION statement. If none is found, control is returned to the target program; otherwise, the sequence of commands within the EXCEPTION are read and executed in a sequential manner. The DEP continues to process the file until the number of statements given in the table is processed or until a TERMINAL command returns control to the user's virtual terminal. When control is passed to a terminal, the pointer to the last statement read is maintained and an END_TERMINAL statement will cause the system to start processing at the next statement on the file. If the user issues a RUN statement from the terminal, the remainder of the statements in the EXCEPTION block are ignored and control is passed back to the target program.

3.2.3.3.7.2 ACCEPT Statement Processing

The processing involved in the ACCEPT statement is almost identical to that of the EXCEPTION statement. In both statements, the same table and relative record files are used internally. The target machine address or the keyword ALWAYS replaces the exception_name in the table.

If the parameter is a selected_component, the last item in the component is scanned. If it is an integer, the statement map of the program unit is read and the number tested to make sure it is a valid statement number. If either the statement map is missing or the number is invalid, an error message is generated and the ACCEPT statement is ignored. Otherwise, the statement number is converted to a segment number, entry number, and displacement value, the table updated, and the information written to the relative record file.

If the last item is a name, the statement map and symbol map of the program unit are used to validate it as a label. If either map is missing, the name cannot be found on the symbol map or it is not a statement label, an error message is generated and the ACCEPT statement ignored. The method for resolving this is:

1. Find the symbol on the symbol map.

2. Find a reference to the symbol table node in the statement map.

3. Replace the label with the proper statement number.

Once the proper statement number has been supplied, the statement number is converted to a segment number, entry number and displacement value the table is updated, and the information written to the relative record file. ACCEPT statements apply to every occurence of the statement in all tasks. If the breakpoint is currently "set", the breakpoint table is updated with the location of the "ACCEPT" in the ACCEPT/EXCEPTION table.

3.2.3.3.8 Breakpoint Processing

When a breakpoint occurs, execution of the target program is halted, the target program is halted, control is passed to the debug executive, and the segment number, entry number, and displacement value are sent to the debug executive. When an EXCEPTION is raised, an indication of the type of exception is also supplied.

Once processing has passed to the debug executive, the field containing the first byte of the first machine instruction in the breakpoint table is scanned to obtain the statement number and see if the breakpoint was explicitly "set". If the breakpoint was not explicitly "set", the debug executive test to see if a variable breakpoint has occurred. If so, the debug executive:

1. Generates a message giving the location of the breakpoint.

2. Generates a message giving the name of the variable and the new value.

3. Proceeds to ACCEPT statement processing.

Otherwise, control is passed to the target program. If the breakpoint has been explicitly "set", a message is generated giving the location of the breakpoint and the executive proceeds to ACCEPT statement processing. ITEM

If the breakpoint's entry in the breakpoint table indicates there is an ACCEPT statement, the statements are read from the relative record file and executed until either the number of statements in the table or a statement transfers control elsewhere. When the final statement has been processed, any commands associated with an "ALWAYS" entry are processed. If the "ALWAYS" is processed to the last statement, control is returned to the target program.

If there is no ACCEPT for a given breakpoint: in debug mode, control is passed to the user's virtual terminal; in batch mode, control is returned to the target program. Breakpoint processing interactive mode is terminated via either a RUN or ABORT command. Passing control back to the target program requires the debug executive fetch the original code for the first statement from the breakpoint table then place it in its correct place in the target program and place a "reset" interrupt in the first byte of the last machine instruction in the statement.

In processing a "reset" interrupt, the debug executive replaces the original code at the start of the source statement with a "set" interrupt, replaces the "reset" interrupt with the original code, and returns control to the target program.

SECTION 4

QUALITY ASSURANCE PROVISIONS

4.1 Introduction

Testing of the Toolset CPCI shall be in accordance with the schedule, procedures and methods set forth in the following documents:

1. Contractor's Computer Program Development Plan (CPDP)
2. Computer Program test Plan for each tool in this CPCI.
3. Computer Program Test Procedures for each tool in this CPCI.

Testing of each CPC shall be performed at three levels:

1. Computer program unit test and evaluation
2. Integration test, involving all components of the CPC
3. Computer program acceptance testing, involving the APSE

4.1.1 Computer Program Component Test and Evaluation

This level of testing supports development. Each program component of this CPC shall be tested as a stand-alone program before integration with the APSE. This testing shall concentrate on areas such as the following:

* Text editor command language

* Binder commands

* Debugger interactive commands

An overall system test plan and schedule shall identify the parts of the system that must be available for testing of each component.

Test results shall be recorded in informal documentation; formal test reports are not required.

4.1.2 Integration Testing

This level of testing supports integration and prepares for acceptance testing of each CPC. Each CPC shall receive separate integration testing, using available components of the complete system. Testing at this level shall follow the approved test plans and procedures. Formal test reports are not required.

4.1.3 Formal Acceptance Testing

This testing assures that the Ada Integrated Environment system and its constituent CPCs conform to all requirements in the Type A and B5 specifications. A formal test plan and test procedures shall be generated and used to insure satisfaction of all requirements. Acceptance tests shall be defined to incrementally test major functional capabilities of the three tools in this CPCI: the text editor, the binder and the interactive debugger. Acceptance testing shall be witnessed by the Government. Test results shall be documented in accordance with the Computer Program Development Plan and Computer Program Test Plans, and delivered to the Government with final system documentation.

4.2 Test Requirements

Unit testing and integration testing shall be performed using the developed computer program components and needed drivers. While testing shall not use formal test plans, testing shall keep the final acceptance tests in mind. Unit tests and integration tests consist primarily of an exercise of each specified feature of each computer program component. Each language feature of the control languages for the text editor, binder and debugger shall be separately tested.

The test plans for each CPC shall specify the completeness of testing to be achieved by describing all logical paths through the code of each developed program and identifying testing conditions that will traverse the appropriate paths.

4.2.1 Rehosting tests

Parallel sets of tests at all levels shall be developed for each CPC to be run on the IBM 370 and the Interdata 8/32 host systems. Components of the 370 version may be used to simulate or provide drivers for components not yet rehosted on the Interdata 8/32, during unit testing.

4.2.2 Performance Requirements

The performance of each system component shall be measured in terms of its use of host system resources and in the efficiency of the software products it generates.

The Government shall specify the machine and operating system configurations for the initial Ada Integrated Environment host systems. Acceptance test plans shall specify CPC performance requirements in terms of processing speed and memory use in these host systems. Interactive response time criteria and criteria for efficiency of database mass storage use shall be specified for appropriate system components.

Acceptance test plans shall specify performance requirements, in terms of processing speed and memory utilization on the host systems, of selected test programs generated from input Ada source text by the Ada Integrated Environment compiler and software toolset.

4.3 Independent Validation and Verification

An independent validation and verification (IV&V) contractor, if one participates in the Ada Integrated Environment program, may perform independent testing of the Ada compiler using any of the tests descibed above or additional procedures.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

8